# Visualizing Software Architecture Evolution using Change-sets

Andrew McNair, Daniel M. German, and Jens Weber-Jahnke
Computer Science Department
University of Victoria
Victoria, Canada
{amcnair,dmg,jens}@cs.uvic.ca

## Abstract

*When trying to understand the evolution of a software system it can be useful to visualize the evolution of the system's architecture. Existing tools for viewing architectural evolution assume that what a user is interested in can be described in an unbroken sequence of time, for example the changes over the last six months. We present an alternative approach that provides a lightweight method for examining the net effect of any set of changes on a system's 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.*

## 1 Introduction

Software evolution refers to the behavior of software systems as they change over their lifetimes. In a process akin to biological evolution, it is now generally agreed that software systems must continually adapt to remain useful. There are three main groups interested in this 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 previous progress to help plan future development work.

Although any artifact generated during the development process may provide insight about the evolution of the system, the majority of research has focused on examining the artifacts stored in a software repository, and their associated metadata (see [15] for a comprehensive review of this type of research). A common technique for helping to summarize the massive amounts of data stored in a software repository is to visualize it, with the varied motivations for studying software evolution resulting in a diverse set of approaches to visualizing the data [23].

We believe that existing approaches to architecture evolution visualization suffer from a lack of flexibility in how they allow users to customize their view. Generally, these tools allow users to filter their results by time period, for example showing all changes to the architecture within the last six months, and by level in the architecture's hierarchy, for example showing all changes that occurred within a specific package. This type of customization is sufficient to show a high-level overview of how the architecture has changed over time; however, we believe greater flexibility is necessary to deal with the broad range of questions that developers, researchers, and managers may want to answer.

As with most other approaches, our visualization shows the evolution of the semantic entities that make up the architecture of the software system being studied (such as the packages, classes, and methods). As with some other approaches, we provide the ability to understand how particular changes have affected the evolution of the system, and to discover further details of these changes (including who made the change, and when). In what we believe is a novel contribution, we also allow users to view the net effect on the system's architecture of any set of changes they consider logically related. We call this set the *change-set*.

The structure of the rest of this paper is as follows. Section 2 introduces our method for visualizing architecture evolution. Section 3 presents Motive, a prototype tool we have created that implements this visualization. Section 4 reports the case studies we conducted to evaluate whether our approach has merit. Section 5 describes related work. Finally, Section 6 concludes the paper and discusses future work.

## 2 Modeling software architecture evolution

Most tools that explore the evolution of the architecture of a system have taken a time-based approach which

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.

shows how the state of the system (or a sub-system), has changed between two moments in time. These moments might be releases of new versions of the system, two dates that the user is interested in, or two particular modification records (MRs, equivalent to a logical commit). In combination with the metadata available from the version control system, these tools can be used to answer questions such as "who made an architectural change during this period?", "what particular MRs changed the architecture during this period?", and "what has been added during this period?"

However, time-based approaches do not provide good support for evaluating the impact of several MRs when these MRs are not sequential in time. For example, in the case where three MRs ($a, b$ and $c$) were applied (in that order) to the system, time-based approaches would have difficulty showing the user the impact of only $a$ and $c$ on the architecture while ignoring the effects of $b$. Our approach is geared towards answering these types of questions, when a user says, "I want to know the impact of these specific MRs and I do not want to consider the impact of the other MRs."

Rather than using a time-based approach we allow the user to examine the effects of a **change-set**, which we define as being a subset of the MRs of a system. There are no restrictions on what MRs can compose a change-set; it can be built by enumeration (listing each MRs in the change-set), or by stating a property that its MRs should satisfy. These properties come from either the MRs' metadata, such as "MRs by a given developer between these dates," or synthesized information, such as "MRs that represent defect fixes" or "MRs that involve refactoring."

We indicate the architecture evolution of a system in terms of the evolution of the entities of the system that define its architecture, such as packages, classes, methods, functions, etc., and the relationships between those entities. The period of evolution we are interested in takes place from the *starting version* of the system, immediately before the first MR in the change-set, through the MRs in the change-set, and up to the *final version* of the system, immediately after the last MR in the change-set. Our diagrams show the architecture of the system over the period of evolution, with each entity/relationship annotated to show how it was affected by the MRs in the change-set. The simplest type of annotation assigns one of the following states to each entity/relationship:

**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, affected by at least one MR (and are not added, deleted, or phantom).

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

These states can also represent the moving of an entity. This can be shown by changing the entity's metadata (to reflect its new location), deleting any existing relationships dependent on its old location, and adding these relationships to its new location.

## 2.1 Architectural impact view

We show the impact of the change-set on the system's architecture using what we term an *architectural impact* view. An architectural impact view is based upon an architectural diagram (such as a UML or E-R diagram) that has been enhanced to depict the impact that the change-set had on the system. Neither UML nor E-R diagrams prescribe the use of color; for that reason we have chosen different colors to show the impact of the change-set on each of the elements of these diagrams (other visual attributes could be used, if desired).

In an architectural impact view we draw each entity/relationship in a color that corresponds to its final state (saturation can be used to depict the "amount" of change an entity/relationship suffered–such as the proportion of MRs in the change-set that modified it). Other annotations methods (such as overlays) can display more information about the list of events that occurred to each entity/relationship (such as changes in the metadata).

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 users (Author1, Author2, and Author3) that have made changes (a total of 6 MRs) to the system between the system's two releases (R1 and R2). The entities in our example system and their changes are depicted in Figure 1. Figure 2 shows the state of the system at R1 and R2. In this small example we are using simplified relationship diagrams which display entities and the existence of relationships between them. Because in this example we are focusing on the entities, the types of relationships are not specified, nor do we track changes to relationships except when an entity is added or deleted. 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).

Figure 3 shows an architectural impact view of our example system, where the change-set includes all MRs. We

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.

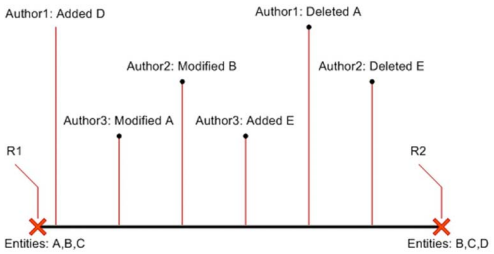


**Figure 1. Evolution of an example system**

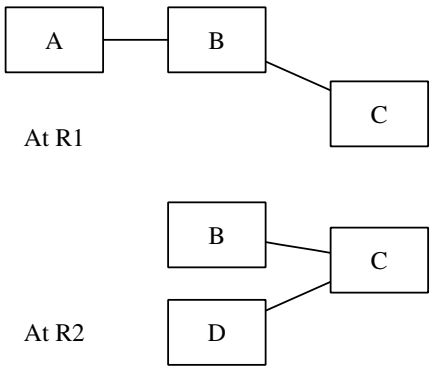


**Figure 2. Simplified relationship diagrams for the example system at R1 and R2**

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*[1]. This view provides a comprehensive overview of what architectural change occurred between R1 and R2: $A$ was deleted (black), $B$ was modified (yellow), $C$ was unchanged (grey), $D$ was added (green), and $E$ was added and then deleted, and 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.

We can also create different change-sets and show their impact. The simplest possible change-set includes only one MR. For example, Figure 4 shows the effect of the fifth MR ($A$ is deleted)[2].

Figure 5 shows the impact of the change-set comprising the MRs of author Author2. Author2 modifies $B$ (MR #3)

[1]For those readers viewing a black and white version of this document, we invite you to see the images in the electronic version where they are displayed in color. For your benefit we also describe in footnotes the colors used in the diagrams.

[2]$A$ is black and the rest of the nodes are grey.

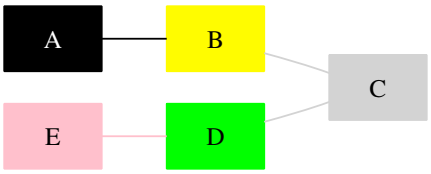


**Figure 3. Architectural impact view comparing R1 to R2 (change-set is equal to all the MRs in between both releases)**

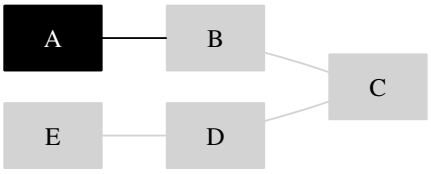


**Figure 4. The impact of the fifth MR. In this case the change-set and the total set both contain only one MR. $A$ is deleted**

and deletes $E$ (MR #6)[3]. This diagram does not reflect the current architecture at the time of R2 (see Figure 2), only its evolution in response to the MRs authored by Author2. For instance, it does not show that $A$ is no longer in the system ($A$ was not deleted by Author2).

## 2.2 Computing the impact of a change-set

The combinatorial explosion of possible change-sets precludes the possibility of pre-computing their impact. Instead, we have developed a lightweight algorithm that quickly computes the impact of a change-set selected by the user. This approach works over the *period of interest*: the period of time between the earliest and the latest of the MRs in the change-set. We refer to the MRs in this period of interest as the *total-set*. The change-set is a subset of the total-set.

[3]$A$, $C$, and $D$ are grey, while $E$ is black and $B$ is yellow.

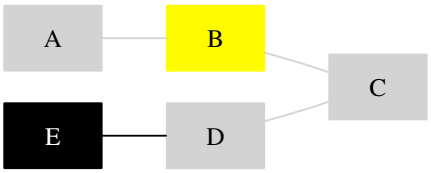


**Figure 5. The impact of the changes of Author2**

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.

Evaluating the impact of some MRs while ignoring others is not trivial. An ignored MR might add, change, remove, rename, or move entities that are subsequently altered by a later MR. Although we are only interested in the effects of the MRs in the change-set we need to be aware of the effects of some other MRs. For example, we need to know if an entity or relationship is added or deleted in any MR in the total-set, whether is part of the change-set or not.

From the version control history of the system we have extracted from each MR information about *what* entities and relationships were altered, and *how* (added, deleted, modified, etc.). This operation is done only once per MR.

Our algorithm takes as input the list of MRs in the total set (including the information of *what* and *how* they have changed the system), and the list of MRs in the change-set. Its output is a list of entities and relationships, annotated within an architectural view with a summary of the net effect of the MRs in the change-set.

The algorithm has two stages. It begins by analyzing the state of the system at the starting version and building a "current list" of entities and their relationships, and, for each entity/relationship associating an empty "annotation list" that will keep track of its changes. Now, for each MR 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 (which means the user is interested in the changes made in that MR), 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 we can display to the user all the entities in the system and the relationships between the entities. 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 (renaming an entity/relationship will update its metadata). The second stage of the algorithm inspects the annotation list of each entity/relationship to determine its final state:

- 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*.

The final state of the entity/relationship determines the color used to depict such entity. Using other visual attributes or overlays the annotation list could be used to display other information, such as who performed the changes, when, how many times was the entity/relationship changed, how recently was it changed, etc.

## 3 Motive

To evaluate our method, we have developed a prototype tool that enables us to visualize the evolution of Java systems stored in a CVS repository. We named our tool *Motive*, as one of our main goals is to provide greater understanding of the reasons for architectural drift and evolution. CVS was chosen because it is a popular choice of software repository 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 is far easier to statically analyze than languages that support pointers such as C++.

### 3.1 Preprocessing

We begin by preprocessing the data in the CVS repository so we have the information necessary to compute the impact of the MRs in a change-set. We use a three-step process. In Step 1 we use *softChange* [7] to extract facts from the CVS repository. The output of this step is a database containing information about each MR (the author, log, date, and time), as well as the file revisions associated with each MR. Although softChange extracts information about all MRs in the CVS repository, we only consider changes to the trunk of the CVS repository, ignoring branches. This allows us to simplify our visualization requirements.

In Step 2, for each trunk MR the most recent version of any files modified in that MR are scanned, and the current model of the software system's architecture is updated accordingly. As with [12] we use a scanning and not a parsing approach, giving us the ability to include changes that may have "broken the build." We use *QDox* [20] as a scanner, which gives us information about both entities (packages,

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.

classes, interfaces, methods, and fields), and relationships (specialization and generalization). We also attempt to detect usage dependency relationships; although, because we are not parsing method bodies, we may miss some of these.

In Step 3 we compare the entities and relationships existing in the software system in two consecutive MRs to identify the effect of the latter MR relative to the first. Our implementation is currently more limited than our model. We are not yet detecting the renaming or moving of entities; and consequently, nor are we displaying metadata changes. We plan to extend our implementation to incorporate existing methods for detecting renaming and moving, such as the Bertillonage analysis approach developed by Tu and Godfrey [26].

### 3.2 User interface

Motive can be used to visualize the net effect of a change-set on a software system's architecture, or to export this visualization to more mature graph visualization tools that support GDL or GXL. We are currently using *aisee* [1] for the visualization of large graphs.

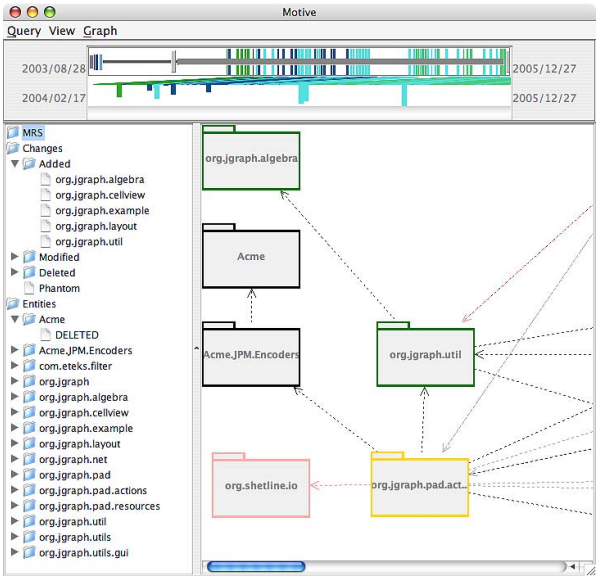


**Figure 6. An overview of Motive**

Figure 6 shows an overview of the three main panels of the Motive tool:

1. 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 upper dates indicate the spread of MRs in the most recent query, in this case the period under consideration goes from 2003/08/28 to 2005/12/27. The lower dates indicate the period currently being studied in the change-set, in this case from 2004/02/17 to 2005/12/27. The user can also "lock" the two sliders together to view the effects of a particular MR.

   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 10 MRs, of those currently under consideration, that have modified the most files. For example, we can see that the MR that modified the most files, shown as the largest MR on the bottom half of the slider, occurred towards the end of the MRs in the change-set. We can also look up the color mapping of the MR (not shown) to determine which author made the change.

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 visualization we developed. Currently these views are limited to UML class and package diagrams.

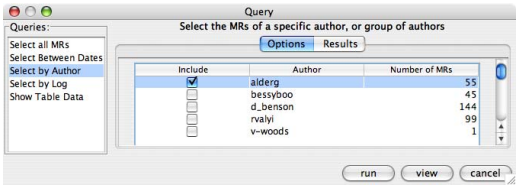


**Figure 7. The Query Dialog**

To make it easy to select change-sets, we have created a few sample queries, as well as a plugin mechanism for adding queries. The only requirement of these queries is that they return a list of MRs, enabling the construction of a change-set. Figure 7 shows an overview of the Query Dialog, which lets the user run one of the sample queries we developed. In this case, the user has chosen "select by author" which lets the user build a change-set from the MRs made by one or more authors.

## 4 Case studies

Our visualization method and Motive tool were designed to summarize information about a software system's evolu-

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.

tion. In order to evaluate their effectiveness, we conducted two case studies. In each case study we asked the same questions about the evolution of the software system under consideration, and tried to answer these questions with Motive.

### 4.1 Questions

The questions we posed were based on scenarios that we consider are typical for developers and their managers, and on typical research questions posed by software evolution researchers. These scenarios were largely created based on plausible use cases rather than empirical data. A notable exception are the scenarios developed by Wu [28] that are based on a survey Wu et al conducted about the concerns of developers and managers [29] .

#### Researcher questions

1. *What packages are highly coupled?* This is a first step in identifying the *evolution-critical* and *evolution-sensitive* modules discussed by Tonu, Ashkan, and Tahvildari [25]. 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 [18]. To identify changes that broadly affected the architecture, and so were probably not anticipated by the original developers, we considered the term "architecturally disruptive" from a number of different perspectives, including the number of classes and dependencies the change added, changed, or deleted.

4. *What packages have been modified by a broad group of developers?* That a module is modified by many developers could be considered a sign of good programming (many developers know these modules and can modify them) or poor design (the need for the module to be modified by many people might hint at a need to split the module). We determine what constitutes a "broad group of developers" from the percentage of authors in the system that have made changes to the package.

5. *Given a keyword, which 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 the change and, indirectly, the purpose of the code. They use the example of the CVS log "added footnote feature" which indicates the corresponding change is related to footnotes and that the modified source code has to do with footnotes. In this example, 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 [27] 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 to consist of 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 module/package, and who made the last modification?* Wu [28] noted the importance of knowing who is making changes to which part of the software. For example, in the case a developer looking for someone to ask for help understanding an unfamiliar package, it is likely a 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 given days globally or in a specific package?* It is often the case, as reported by Wu [28], 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, particularly in certain packages, during the time she was away.

9. *What happened in a particular commit?* Wu et al [29] found that developers will often want to examine a particular commit. This is arguably 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

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.

packages developers are currently working on. As well, there is some evidence that classes changed recently are likely to be changed in the future [9]. We identify the recently modified packages by finding what developers have been active in the last 2 weeks, and, of those developers, what packages they have 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 [8] [28] [24]. 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 last 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.

### 4.2 Systems studied

Motive was used to answer the previous questions during the study of two systems: *JGraphpad* [14] and *SQuirreL-SQL* [22].

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 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.

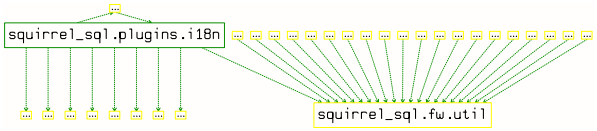


**Figure 8. New dependencies from internationalization**

### 4.3 Results

Motive was able to answer the developer and manager questions posed, and was able to help with the researcher questions, although questions 1 and 3 could only be partially answered. In Question 1, it was possible to detect from JGraphpad's small dependency diagram what the most coupled packages were. However, the diagram of SQuirreL SQL proved too large to easily answer the question. In Question 3, 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 determine the most disruptive changes according to the other criteria specified in the question.

As expected, the ability to create flexible change-sets was useful. This usefulness manifested itself in three areas:

1. *Studying groups of changes based on log entries*

For example, one term that frequently occurs in the logs of SQuirreLSQL is "i18n", a standard abbreviation for internationalization. In fact, 783 MRs have this term in their logs. If 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 8 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.

2. *Examining author changes*

Answering Question 12 requires highlighting the changes of a particular author. During our study of JGraphpad, we selected the changes of d_benson to exam-

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.

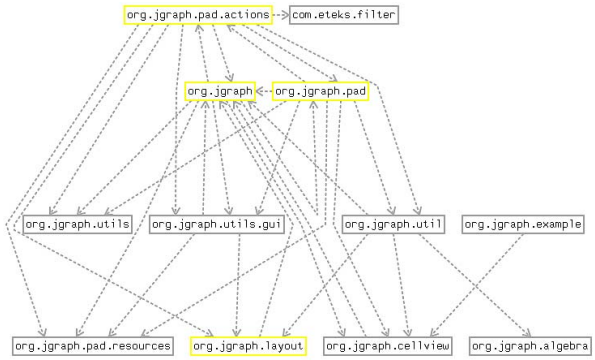


**Figure 9. Modifications made by d_benson in the last 3 months**

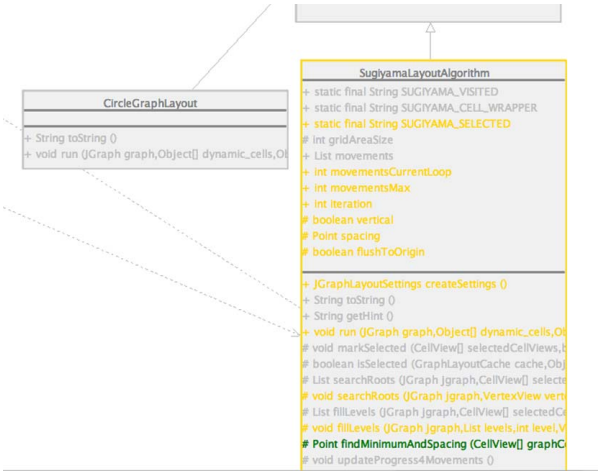


**Figure 10. A highlight of modifications to SugiyamaLayoutAlgorithm in the org.jgraph.layout package**

ine. Figure 9 shows a high-level overview of the effect d_benson has had on the architecture over the last 3 months we have data for. 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 10 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.

3. *Filtering large transactions*

In [31], 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.

## 5 Related work

One classification for software visualization tools was developed by Maletic, Marcus and Collard [17], in which they describe the tools according to five dimensions: i) Tasks: *why* is the visualization needed? ii) Audience: *who* will use the visualization? iii) Target: *what* is the data source to represent? iv) Representation: *how* to represent it? v) Medium: *where* to represent the visualization?

We believe the majority of software evolution visualizations can be grouped into four broad categories according to the target of each visualization. We term these categories: artifact-centric, metric-centric, feature-centric, and architecture-centric. Our work falls into the architecture-centric category, though change-sets may be of use to other visualization methods.

### 5.1 Artifact-centric

These tools are designed to provide a view of how some artifacts stored in the software repository change over time, especially the source code files and the lines of code within the files. Examples of tools that fall in this category include: the visualization of Van Rysselberghe and Demeyer [21], Revision Towers [24], Evolutionary Storyboards [2], Xia [28], and CVSScan [27].

### 5.2 Metric-centric

These tools are designed to provide a view of how some software metrics have changed over time. Many tools include some form of metrics in their visualization, but tools in this category use metrics as the primary target. Examples of tools that use this approach are: Evolution Matrix [16], the work done by Bieman, Andrews, and Yang [3], RelVis [19], the Hierarchy Evolution Complexity View [10], and SourceViewer3D [30].

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.

### 5.3 Feature-centric

These tools are designed to provide a view of how features of the software have changed over time. Because there is no direct way to extract the features from source code, this requires an analysis performed on at least one other data source in addition to the code in the software repository, such as comments in the logfiles, data from a bug repository, or execution traces of a running software system. Feature-centric approaches include that of Fischer and Gall [5], and the work of Greevy, Ducasse, and Gîrba [11].

### 5.4 Architecture-centric

These tools are designed to provide a view of how the architecture of the software has changed over time. There are three main approaches to architecture-centric visualization: visualizing the entire architectural evolution at once, showing deltas of the architectural differences between two releases, and a unified approach that gives both an overview of the architectural change and highlights of specific differences between releases.

Gall, Jazayeri and Riva [6] developed an example of an overview approach 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. This 3D view enables a user to detect the main changes in the evolution of the system, while the 2D view allows the user to view in more detail the changes to subsystems in a release. However, this approach does not allow viewing the effects of specific MRs, or groups of MRs.

GASE, Graphical Analyzer for Software Evolution [13] is an example of an approach to comparing two releases of a software system by visualizing architectural deltas. Color is used to illustrate the change between the two releases, showing what is added, deleted, or common between the releases. Holt and Pak, the authors of GASE, also mention that their visualization approach could be extended to viewing multiple releases by using the color intensity to represent how recently a module or relationship was added or removed. Although we are representing more types of change, our coloring scheme is similar, and we are interested in experimenting with GASE's change in color intensity over time.

One tool that provides both an overview of how the software architecture has evolved and a more detailed description of the changes between particular releases is Beagle [26]. One of Beagle's panels shows a tree view of the system's structure at one particular version. The other shows a dependency diagram of how the software has evolved over a selected number of versions either to or from the version of the system highlighted in the other panel. Color is used to indicate entities that have been added, modified, and deleted, with intensity used to indicate ordinal attributes, such as how long ago an entity was added. Beagle detects the moving and renaming of entities between versions; as mentioned earlier we need to integrate this type of analysis in order to detect metadata changes.

Another technique for viewing both an overview of how software changes over time and a more detailed view of particular times is animation. An example of an animation approach is YARN [12], which uses animation to display the architectural dependency graph evolving over time, and several coloring schemes to emphasize different aspects of the evolution. As with our approach, YARN allows the fine-grained viewing of individual MRs, which are each presented as a frame in the animation. In contrast to our approach, YARN is focused on showing the evolution of the software over unbroken periods of time, not over change-sets.

## 6 Conclusions and future work

In this paper we presented a novel type of architectural visualization that enables the user to view the effects of a particular MR, or a set of MRs, on the architecture of a software system. Our intuition was that the standard methods of filtering by time period or level in the architecture's hierarchy were insufficient for some tasks. We also believed that viewing the impact of change-sets might help with these tasks. Although more evaluation is required, based on two case studies we conducted of the prototype tool that implements our visualization, we believe that our intuition was correct. Visualizing the impact of change-sets seems to have a lot of promise in its ability to help developers, managers, and researchers trying to understand the evolution of a software system.

In the future, we hope to apply change-sets to other methods of evolution visualization, including artifact, metric, and feature-centric approaches. As well, there is a lot of work to be done in improving the implementation of our current architecture-centric approach, and possibly extending it, for example with animation. Another research direction would be to explore the different types of change-sets that can be constructed. Of critical importance in directing this research is conducting empirical studies to determine the limitations users have with current visualization approaches.

## References


[1] AbsInt. aisee website: `http://www.aisee.com/`. [Online; accessed 15 - June - 2007].

[2] D. Beyer and A. E. Hassan. Evolution storyboards: Visualization of software structure dynamics. In *International*


Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.

*Conference on Program Comprehension (ICPC'06)*, pages 248–251. IEEE Computer Society, 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*, page 44. IEEE Computer Society, 2003.

[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 17th IEEE International Conference on Software Maintenance*, page 364. IEEE Computer Society, 2001.

[5] 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*, 16:385–403, 2004.

[6] 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*, page 99. IEEE Computer Society, 1999.

[7] D. M. German. Mining cvs repositories, the softChange experience. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 17–21, 2004.

[8] D. M. German and A. Hindle. Visualizing the evolution of software using softChange. *Journal of Software Engineering Knowledge Engineering*, 16(1):4–22, Feb. 2006. Special Issue of Best Papers SEKE 2004.

[9] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society, 2004.

[10] T. Girba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 2–11. IEEE Computer Society, 2005.

[11] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing software evolution through feature views: Research articles. *J. Softw. Maint. Evol.*, 18(6):425–456, 2006.

[12] 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.

[13] 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)*, page 163. IEEE Computer Society, 1996.

[14] JGraph. Jgraph website: `http://sourceforge.net/projects/jgraph/`. [Online; accessed 15 - June - 2007].

[15] 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.*, 19(2):77–131, 2007.

[16] 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*, pages 37–42. ACM Press, 2001.

[17] 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*, page 32. IEEE Computer Society, 2002.

[18] D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[19] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of the ACM Symposium on Software Visualization*, pages 67–75. ACM Press, 2005.

[20] QDox. Qdox website: `http://qdox.codehaus.org/`. [Online; accessed 15 - June - 2007].

[21] 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*, pages 328–337. IEEE Computer Society, 2004.

[22] SQuirreLSQL. Squirrelsql website: `http://squirrel-sql.sourceforge.net/`. [Online; accessed 15 - June - 2007].

[23] 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*, pages 193–202, 2005.

[24] C. M. B. Taylor and M. Munro. Revision towers. *vissoft*, 00:43, 2002.

[25] 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*, pages 261–270. IEEE Computer Society, 2006.

[26] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, pages 127–136. IEEE Computer Society, 2002.

[27] L. Voinea, A. Telea, and J. J. van Wijk. Cvsscan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 47–56. ACM Press, 2005.

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

[29] 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*, pages 90–99, 2004.

[30] X. Xie, D. Poshyvanyk, and A. Marcus. Visualization of cvs repository information. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 231–242. IEEE Computer Society, 2006.

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

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on January 12, 2009 at 14:38 from IEEE Xplore. Restrictions apply.