EvoLens: Lens-View Visualizations of Evolution Data

Jacek Ratzinger and Michael Fischer
Vienna University of Technology
Institute of Information Systems
A-1040 Vienna, Austria
{ratzinger,fischer}@infosys.tuwien.ac.at

Harald Gall
University of Zurich
Department of Informatics
CH-8057 Zurich, Switzerland
gall@ifi.unizh.ch

ABSTRACT

Visualizing software evolution is essential for identifying design erosions that have occurred over the past releases. Making evolutionary aspects explicit via visual representations can help the engineer identify such hot-spots quickly. One challenge is to provide means for an engineer that allow her to focus on particular software parts. Although many tools exist that provide zooming-in and -out within the hierarchical decomposition of a software system, only very few allow an engineer to view a system through a kind of lens view. Our approach called *EvoLens* is a visualization approach for efficient explorations of evolution data across multiple dimensions. *EvoLens* is based on structural and temporal lens views, a technique similar to fisheye-views. But the graphical representation of *EvoLens* integrates enhanced zooming by navigating through software hierarchies with arbitrary selectable groups of software parts across module or package boundaries. *EvoLens* allows an engineer to define a focal point for the lens view and navigate along the time dimension by user-defined time windows. The comprehension is supported by using color for metrics of the software modules or classes. The *EvoLens* prototype tool has been developed and evaluated on basis of a medium-sized Java application consisting of 580.000 LOC over an 18 months evolution period.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*interactive environments*; D.2.7 [**Software Engineering**]: Maintenance and Enhancement—*restructuring, reengineering*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, evolution measures*

Keywords

software architecture, evolution, information visualization, color, interactivity

1. INTRODUCTION

As software evolves it changes its size, complexity, and characteristics through modifications. The major costs do not arise because of software bugs, but because new and changing requirements lead to adaptations and improvements [16]. As a result, it is important to keep software maintainable to ensure adequate responses to the users' needs. Parnas states that software aging will occur in all successful products [15]. Software passes through many stages during its life cycle. One of the models for the life cycle is the "staged model for the software life cycle" of Rajlich and Bennett [18]. To understand existing systems better, software engineers should be enabled to effectively exploit the information about the software's evolution. Especially the relationships between classes can lead to important findings [8]. All this information about a program may help the developer to build a mental model of its structure.

In this paper, we present *EvoLens* an approach for visualization and navigation of information extracted from the software development process. The data about the evolution is extracted from versioning systems. In particular we analyzed Java programs that are stored and maintained within a Concurrent Versions System (CVS) [1]. We extract the following data: the author of each change to a file, date and time of each change, the files included in a change event, and the package structure of the source code. *EvoLens* works with the following *abstractions*: *time* on the basis of version history data, *hierarchies* on the basis of software structure, and *change couplings* as representation of common change histories of software parts.

As *visualization models* we use nested graphs to visualize software structure on different levels of granularity. Further, we use color to represent the size of software parts by mapping particular class metrics to a color scale ranging from light yellow to dark red. We, thereby combine structure and size and growth metrics into our graphical representations: class metrics can be compared by the engineer on basis of their color and their structural relationships to other classes or modules. Additionally, one major visualization model is a lens view that is significantly different in its composability to what has been presented in literature so far. Software parts from any decomposition level can be selected and shown in a lens view, which is a focus+context approach. The lens view zooms in from a module perspective to a class perspective enriching the level of detail to inter-class relationships. In each lens view, the engineer selects a so-called *focal point* that defines the center of consideration. Each *EvoLens* vi-

SOFTVIS '05 Saint Louis, Missouri, USA

sualization uses this focal point as starting point and all relationships following from this focal point to other modules or classes are represented. This is realized by pulling up class-level change dependencies onto the higher level views. This technique drastically reduces the amount of graphical elements to be visualized. It allows us to show both *inter*- and *intra*-class evolution metrics.

EvoLens further provides facilities to generate user-defined projections of interrelated software parts that can be part of completely different modules, into one picture. This enables an engineer to visualize important relationships across module boundaries. We support software engineers by providing navigation facilities beyond panning and zooming. In *EvoLens* it is possible to move the focal point to any module within the system and additionally on every level within the containment hierarchy. Additionally, we also provide navigation in time. The user may move around in history interactively and get the dependencies between classes displayed for the selected time frame. That allows building a mental model of the changes appeared during the development and maintenance phase of the software structures.

The remainder of this paper is organized as follows. In Section 2 we describe how information is collected from configuration management systems. Section 3 presents the types of visualizations *EvoLens* provides together with a brief review of techniques for presentation of complex information spaces. Section 4 describes the extended features of our visualization tool. Section 5 evaluates our evolution analysis approach in combination with *EvoLens*. Section 6 discusses related work and Section 7 highlights important results and indicates some future work.

2. EXTRACTING CVS INFORMATION

CVS allows handling of different versions of files in a cooperating team of developers. As CVS logs every action, it provides the necessary information about the history of a system. The log-information—pure textual, human readable information—is retrieved via standard command line tools, parsed and stored in a relational database [4]. Following the import of the logs, the required evolutionary information is reconstructed in a post processing phase.

Log groups L_n are sets of files which were checked-in into the CVS by a single author within a short time-frame—typically a few minutes. By applying a dynamic time-frame, larger check-in transactions can be captured as well. Following, log groups are then used in the evaluation of logical coupling between software entities. We refer to logical coupling as: *Two entities (e.g. files) are logically coupled if a modification to the implementation affected both entities over a significant number of releases* [7].

The degree of logical coupling between two entities a, b can be determined easily by counting all log groups where a and b are members of the same log group, i.e., $C = \{\langle a, b\rangle | a, b \in L_n\}$ is the set of logical coupling and $|C|$ is the degree of coupling. In the next reconstruction step the file size history is recovered. Since CVS records the number of lines added and lines deleted for each check-in of each source file, their values are summed-up on a per file basis. This information is used as source for size and growth metrics. And finally, the module structure is reconstructed from the path information provided for each file in the repository and listed in the logs. A more detailed description of the extraction- and import-process can be found in [4, 22].

3. VISUALIZING DEPENDENCIES

We aim to support the software engineer when s/he has to learn and understand large legacy software. As legacy systems are often sporadically documented, we incorporate historical data as another source of information into the reverse engineering process. With the help of *EvoLens* the developer obtains deeper insight into the evolution and maintenance process of the analyzed software. Research questions such as *"What are the common work patterns within the development process?"* or *"How are inner class metrics about size related with inter class couplings?"* can be addressed with our approach.

The extracted dependencies between programming entities like classes are depicted by utilizing graphs. The nodes, which describe classes, are connected with each other based on the historical coupling. The thickness of edges between classes represents the strength of the relationship. The more often classes were changed together the stronger is their coupling. However, the resulting graph can become very large. Therefore, we need strategies to support software engineers in handling such large information spaces. Our idea is to group the classes based on structural information and to filter important parts of the graph by using focal points.

While examining the system for common change patterns the attention is drawn towards the modularity of applications. Maintainability can be improved when the system is well composed of self-contained components. An ideal situation would allow changing each component independently of others. If changes must be propagated, the smallest possible set of components should be involved. As a result we organized the graph according to the module structure of the application. The resulting graph groups files, which contain classes according to their package membership.

3.1 Lens View

The lens-view allows an engineer to focus on a particular software part (e.g. *ROOT/chkclass* or *ROOT/jvision/main*). This module is shown in full detail and it is described as surrounding rectangle. The directly included classes and submodules are drawn within this box. On the one hand submodules are depicted as rectangles too. On the other hand, classes are represented through ellipses. Our technique displays a hierarchical tree as a nested graph. The leaf nodes of the module hierarchy are classes. Modules build up the nesting levels of the grouped graph. Figure 1 and Figure 2 describes the correspondence between a hierarchical structure and its nested graph representation. The grey box of Figure 1 represent the focal point for the nested graph visualization. We focus on module *chkclass*, which is shown as surrounding box in Figure 2. Within this module the submodules at Level 3 are incorporated as rectangles. Classes included in the subhierarchy of module *chkclass* are projected on this plain containing *chkclass* and its children *Attic*, *jfolder*, and *dicomsend*. If *chkclass* would contain classes itself they would be directly displayed within the box of this

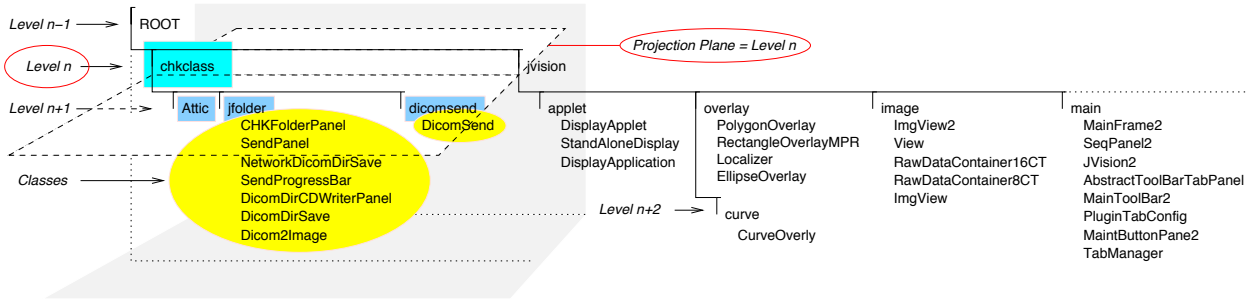

Figure 1: A Java package structure

module. Other classes down the hierarchy are projected on the submodules (e.g. *CHKFolderPanel* on *jfolder*).

Based on nested graph representations of module structures we visualize evolution data. For the evolution of software items we define *coupling* as the relationship based on common changes. In *EvoLens* this coupling information is interweaved with the hierarchy information of the module structure. Classes that are strongly coupled with each other are visualized together with their module decomposition. Edges of different thickness connecting ellipses, which represent classes, give an impression on the common change patterns. An interesting aspect of coupling is the distinction between internal and external. We define *internal coupling* as a dependency that happens between classes in respective parts of the system; e.g. the relations between classes of a single module and its submodules are defined as internal couplings. The connections between classes within this module and any other part of the software (i.e. another module or another subsystem) are considered as *external couplings*.

For the module in the focal point, we paint inner couplings and also couplings to other external modules. For external modules we limit couplings to the ones with the focused module. Hence, we build a boundary of couplings related to the focused module. We get the impression of a magnifying glass, where only the focal point is emphasized. Additionally, the couplings outwards of this focal point are visible. But the structure of the surrounding modules is depicted in the same fashion as the focused one.

EvoLens is not the first tool to leverage hierarchical visualizations. In earlier work, researchers have presented views for hierarchical-organized, two-dimensional graphs [3]. However, our nesting level and the handling of leaf nodes differ. The nested graph does not depict the entire package hierarchy. In every step just two levels are shown. The outer box of the focal point describes a package, which is selected by the user. Within the box the direct children are included. Thus, we assemble classes as members of the package and the subpackages on the next lower level. All classes below these submodules are projected into these submodules regardless how deep they are located in the hierarchy. In Figure 2 we show how classes are projected onto the nested module graph. The reference information was taken from Figure 1. We avoid too many levels of the hierarchy that may overload the graph because of the density of the evolution data.

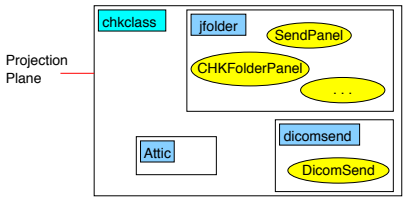

Figure 2: Nested graph representing part of a Java package structure

The graph view addresses the need to describe orthogonal information. The grouping and boxing of classes within modules represents the structure of the software system. The display is recursively broken up into rectangular areas, representing the nested modules. Classes depicted through ellipses are projected onto the nesting structure. The evolution data as second information source relates the classes with each other. Thus, the edges connecting the elliptic nodes concern logical coupling between files. As a third dimension within the visualization color is used to describe the size of each class. The more lines of code each class contains at the end of the given time frame the more reddish the ellipse is.

3.2 Interactive Visualization

Based on the lens-view we provide navigation facilities for an effective exploration of the analyzed software. We depict the coupling of classes; the coupling between modules results from the coupling of its member classes. However, the developer may select the degree of the details he is interested in:

- One can interactively decide which module histories to inspect. This is important since we just show a rather detailed part of the entire information space. Every module on every hierarchical level is reachable through interactive navigation. The focal point may be moved to any sibling module.
- One can determine the strength and number of couplings to see in one picture. He can define the threshold value beyond which couplings are shown.
- The user of *EvoLens* does not have to switch to another view to navigate in time. He can drag the frame

over the time line and adjust the size of the time frame itself. The graph responses interactively to these adaptations.

4. EvoLens

This section describes the capabilities *EvoLens*, and provides some examples. The screen shots are taken from an industrial case study, a Picture Archiving and Communication System (PACS) in the medical domain. The PACS includes a viewing workstation, which supports concurrent displaying of pictures as well as accessing different image sources. The images are acquired from different modalities such as magnetic resonance, or ultrasound scanning and saved in distributed archive storages. The software is implemented in Java. The information of the whole application is maintained with the help of CVS. The development of the software started in 1998. For the visualization examples we used 18 months of the development history. At the end of the observation period, PACS was composed of more than 580 000 lines of code and contained more than 6000 classes.

4.1 Color indicating Class Metrics

We decided to use color to indicate metrics about the evolution of single classes. The basic idea is to color the elements according to their attributes' values. *EvoLens* can indicate three size metrics: The size of each class at the (a) beginning or (b) end of the time window for couplings, or the (c) growth of classes during this time period. This is achieved by mapping size metrics of classes to a color scale from light yellow to dark red (, , , , and). The advantages are that size and growth measures are visualized together with system structure and evolution data. Further, numerical values are mapped to colors, so that the process of comparing these values becomes a perceptive task. The user can visually compare different class metrics based on their coloring. Important events such as a rapid change of size can be easily spotted by navigating over the time line.

Our approach is entirely based on historical data. No source code has to be parsed. The evolution information enables reasoning about the architecture of the system and its development process. We extract all necessary data by parsing change event logs. All change logs declare how many lines of code where added and deleted during the intervention of the programmer. Hence the actual number of lines of code can be determined for every step in time within the software maintenance. For consistence reasons we visualize the size of classes at the end of the observation time frame.

4.2 Multi-dimensional Visualization (Structure and Evolution)

Our visualization framework was designed to enable software engineers to browse the evolution of large systems. *EvoLens* integrates different dimensions of information. We use a combination of structural information enriched with evolution data to provide better understanding of the development process of large software.

Figure 3 shows an example of coupling visualization based on the industrial case study. The focal point (fp) of this figure is on the package *jvision*. The coupling level, which is described in Section 4.7, is set to 10. The time frame

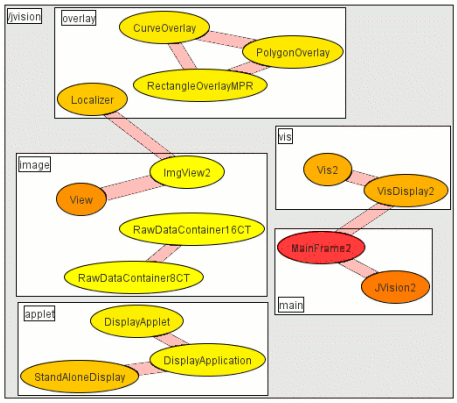

Figure 3: fp=*jvision*, coupling-level=10, time-window=04.2003-09.2004

describing the inspection period for the coupling and other metrics is set from April 2003 to September 2004. The abilities of *EvoLens* to move the time window is described in detail in Section 4.8.

Several aspects of the software can be extracted from Figure 3: The focal point displayed at the top represents module—in Java package—*jvision*. This module is further divided into submodules. All classes displayed for package *jvision* are included in its submodules. Thus, the user has an impression how the system is structured, how many modules are related with each other, and if the modules are further divided into smaller units.

In addition to this structural information, the image describes the evolution of the software. It shows which parts were changed at the same time by the programmers of the development team. This provides hints for further maintenance. This coupling describes implications like: "If class *MainFrame2* has to be changed, then consider also treating other classes like *JVision2, VisDisplay2, etc.*. Do this because they have typically been changed together.

Within the high level picture of the package *jvision*, an interesting phenomenon can be recognized: *jvision* has no external couplings which satisfy the lower bound constraint defined by the user. Hence the outer border of *jvision*, marks the border of the display itself. In the following parts, we will adjust parameters within *EvoLens* to see whether *jvision* really has no couplings.

4.3 Folded Gross Structure

Often it is important to get a coarse grained picture of large software systems. Then, not the couplings between classes are of main interest, but the relationships between entire modules. In *EvoLens* such folded views as depicted in Figure 4 provide a good general map of the system. Within the gross structure only the strongest couplings between modules are shown. Classes are not displayed, but instead all classes are denoted through empty circles within modules. The user may be interested in the individual classes that take part in the overall coupling. In that case s/he can unfold the graph and its parts again. The unfolded total

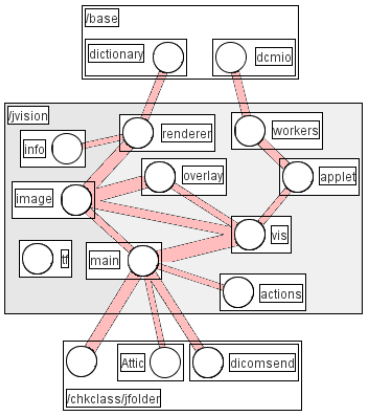

Figure 4: coupling for folded module *jvision*

graph is cluttered, so the user has to reduce the coupling strength and navigate to the points of interest. Through unfolding it is possible to get a fine grained view with all involved classes. Every submodule and every module can be unfolded on its own. For example, it is possible to unfold the focal point to see all the internal couplings but leave the external modules folded. Then only the strongest external couplings are shown. An example for such a view is represented by Figure 5. The user may decide on the balance between the clarity of the image and the detail detailing of the class level.

Folding lays the emphasis on modules. In fact coupling of modules is based on classes, which is indicated through empty circles in Figure 4. The strongest couplings between classes are projected on module level. In Figure 4 the focal point is laid on package *jvision*. Many submodules (e.g. *main*, *vis*, *image*) are displayed. We may easily spot how the internal structure of module *jvision* is build up.

4.4 Selective Coupling

Since module boundaries are sometime too restrictive for in depth inspections, we decided to incorporate the visualization of individually selected sets of classes. The user can mark some classes during the inspection of modules and let *EvoLens* show the evolutionary relationships for the selected set of classes. *EvoLens* identifies the top level modules containing the classes of interest and visualizes them together with the coupling structure. As a result the user can start navigating through the system or select any other arbitrary set of interesting classes.

For Figure 5 we selected four classes: *MainFrame2*, *VisDisplay2*, *ImgView2*, and *Localizer* of Figure 3. These classes are the ones that build up the coupling between submodules. In Figure 5 we folded all modules except for the focal point. By utilizing the features of *EvoLens* we wanted to find out whether these classes are really just related two by two. We set the coupling intensity to a low level to receive also the weak connections. Now we realize that all four classes are related with each other, even though not very strongly. Furthermore, Figure 5 indicates that these classes were changed together with classes of package *jvision*. The

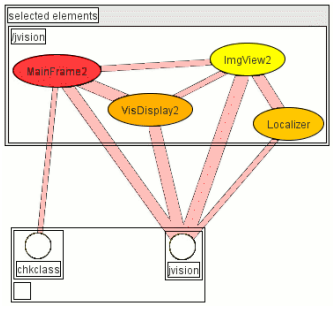

Figure 5: fp=individual selected classes, coupling-level=10, time-window=04.2003-09.2004

classes of module *jvision* involved in the coupling of the set of selected classes are displayed in Figure 3. Additionally to these interesting findings, Figure 5 shows that class *MainFrame2* has weak coupling with classes of module *chkclass*. With the help of this feature the user can select classes from all over the software system and inspect their coupling.

4.5 Zooming through Module Hierarchy

EvoLens is capable of describing two levels of a class hierarchy. For each module all directly included classes and all submodules on the next level are displayed. Within the submodules, classes are directly included as if there were no further levels down the hierarchy. Classes with strong coupling that are located within the sub-hierarchy of submodules are projected onto the submodule level. *EvoLens* allows one to step up and down the hierarchy interactively. The lens view shows one particular module in detail. Within this module the next level of submodules can be directly reached. Thus, the user of *EvoLens* steps down to one of the displayed submodules. In the same manner s/he can step up the module structure. On every step the evolutionary coupling is interactively displayed.

When the level within the hierarchy is changed, also the basis for coupling computation changes. As a result, when zooming in into one submodule new but weaker couplings can be displayed that were filtered on the higher level. When navigating up the hierarchy, weak couplings on lower levels may lose importance, and are omitted given the coupling threshold of the upper layer.

Figure 6 describe the zoom-in into module *jvision/main* visualized in Figure 3 with the focal point *jvision*. After the zoom-in into module *main* the coupling intensity was set to a lower threshold, because the overall coupling strength of this module is on a lower level. Nevertheless, the class *MainFrame2* of module the focused module *main* is still related with other modules of the software. Now this class is related with two other classes of module *jvision* beyond *main*. Many classes of module *main* are involved into the couplings.
There are many external couplings too. In Figure 6 a new module called *chkclass* is related to the classes of the focused module *main*. These coupling between *chkclass* and *main* is also detected in the gross structure of Figure 4. The new focused image shows that the following classes build up the

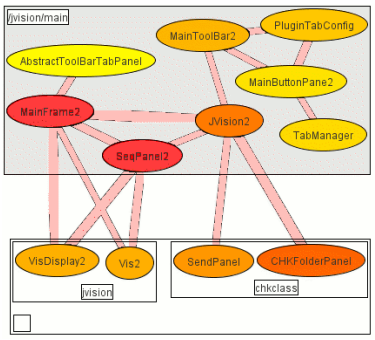

Figure 6: fp=*jvision/main*, coupling-level=10, time-window=04.2003-09.2004

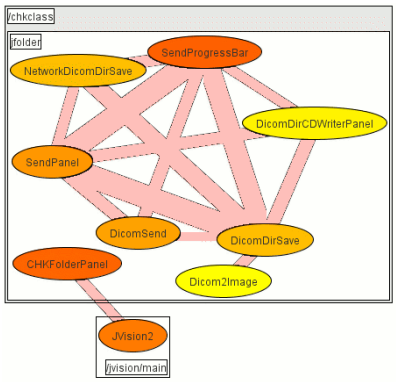

Figure 7: fp=*chkclass*, coupling-level=10, time-window=04.2003-09.2004

intermodule coupling: *JVision2*, *SendPanel*, and *CHKFolderPanel*.

4.6 Navigation between Modules

As well as navigating through the hierarchy, the user of *EvoLens* can navigate to sibling modules and submodules. This horizontal navigation incorporates extended "panning" into our visualization framework. Thus, *EvoLens* provides navigation not only vertically but also horizontally. Interactively users can move the magnifying glass over a related module. Then this module becomes the new focused one. Such a transition takes place from Figure 6 to Figure 7.

In Figure 7 we selected module *chkclass* as the new focal point. The classes connecting *chkclass* and *main* are visible in both figures. However, only two of them (i.e. class *CHKFolderPanel* and class *JVision2* have still coupling displayed in Figure 7. The other couplings are filtered out in order to maintain the clarity of the resulting image. By adjusting the coupling intensity, the user of *EvoLens* has the opportunity to increase the amount of displayed couplings. However, when setting the coupling threshold very low the picture becomes often crowded. Before the navigation to the sibling module *chkclass* classes *SendPanel* and *CHKFolderPanel* of module *chkclass* are connected to class *JVision2* in Figure 6
In Figure 7 most couplings of package *chkclass* are based

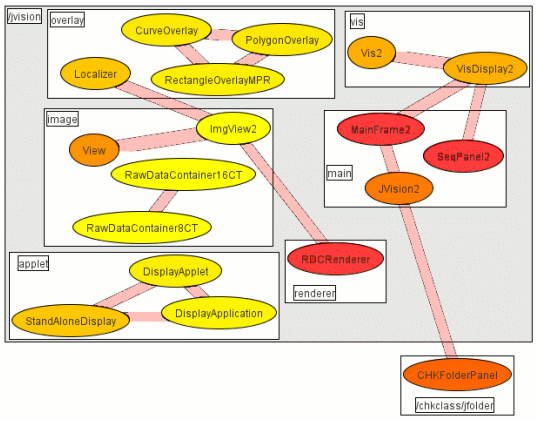

Figure 8: fp=*jvision*, coupling-level=15, time-window=04.2003-09.2004

on classes down the hierarchy. As a result, we see a very dense graph of internal couplings within module *chkclass*. With the help of extended panning to sibling modules and extended zooming through the module hierarchy the entire software system can be explored using *EvoLens*.

4.7 Coupling Intensity

Users of *EvoLens* may decide to navigate through the structure and decide which part of the system they are interested in. On each step only the strongest couplings are depicted. To find a good balance between clarity and information details, the user can individually adjust the lower threshold of the visualized couplings. So she can decide to inspect only coupling with strength higher than any number of common changes, or to see the entire coupling spectrum. However, a view with all couplings shown for a selected module often overloads the display. Classes and modules overlap or cover each other. In such a dense picture the historical couplings can not be easily spotted.

Based on the settings of Figure 3 we instructed *EvoLens* to set the lower bound of couplings to 15. Figure 8 shows the resulting image. For the visualization of the evolution information of the industrial case study, we extracted the historical data from CVS. The data of each change to all classes during the entire lifetime of the system were processed. Then we computed which classes where changed together. The mechanism for this data condensation is described in section 2.

If some files have strong coupling but are on a very low level in the class hierarchy, they are still displayed in visualizations of the higher level packages. This is accomplished by the projection of leaf nodes onto the lowest visualized level. Thus, the depicted submodules contain all classes of their own, their submodules and so on. The strongest couplings down the hierarchy are always presented regardless their exact location down the visualized sub-hierarchy.

4.8 Navigation in Time

Evolutionary couplings are measured on a time frame given by the users of *EvoLens*. For example, the software engineer

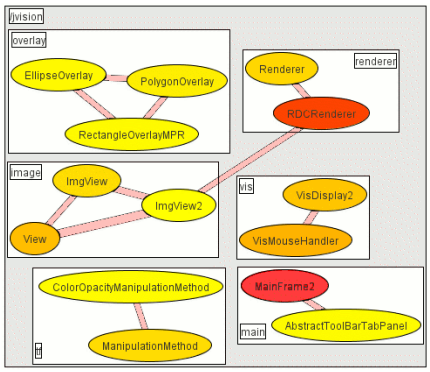

Figure 9: fp=*jvision*, coupling-level=10, time-window=04.2003-12.2003

may be interested in the coupling of a module through the last six months. With the help of *EvoLens* he can set the desired time frame and interactively retrieve the coupling information. To realize the lens view for timely data, we use the Normal Distribution $N_{\mu,\sigma}$ in our weighting function for coupling data. The middle of the time frame t_0 specifies the mean $\mu = t_0$ and the width of the time frame Δt specifies its variance $\sigma = \Delta t$, respectively.

By navigating through the time line software engineers adjust the position of the time frame. They obtain immediately the image of the coupling for the selected period. Figure 3 shows the coupling of module *jvision* within the last eighteen months. In contrast, Figure 9 describes the first half of the eighteen months. So we see how the relationships between the classes evolved. In Figure 9 no coupling can be detected between class *MainFrame2* and class *VisDisplay2*. At the end of the eighteen month period these two classes are strongly related.

The coloring provides additional hints about what happened during this time. In contrast to *MainFrame2*, which was already large at the beginning of the selected time frame, *VisDisplay2* has much more lines of code at the end of the observation period then in the middle This fact is visualized in Figure 9. *MainFrame2* was already large at the beginning of the time frame and remained almost constant. Nevertheless, the strong growth of classes is still an alarming sign, because the strong coupling renders the replacement of the large class or its decomposition into smaller pieces more difficult. The relationship of *View* and *ImgView2* is also striking. Although *View* grew by more than 300 lines of code, *ImgView2* remained almost constant at 150 lines of code. This information is obtained from the coloring of the classes. Although these two classes have strong coupling only one grows whereas the other remains equal in size and is continuously modified. Thus, *ImgView2* is instable, which is a sign of design erosion.

5. EVOLUTION ANALYSIS WITH EvoLens

To support developers in understanding software development dynamics it is critical to effectively structure historical information. Understanding a software system is often difficult due to the overwhelming information mass. Recent studies suggest that programmers use both top-down and bottom-up techniques for the maintenance of software [21]. In our previous work [8] we presented an approach for software evolution analysis. This analysis included several steps combining specific metrics extracted from versioning systems. However, we just presented manually drawn images for a better understanding. In this paper, we present *EvoLens*, a full-fledged visualization and navigation tool. It bases on the idea of logical couplings—evolutionary change dependencies between software parts—but allows an engineer a much more effective exploration of a software system. Release histories contain a wealth of information about the software structure. *EvoLens* leverages such release data to support software maintainers in their daily work through interactive navigation facilities.

In our work on Relation Analysis [8] we measured dependencies and interrelations of classes affecting the maintainability of an object-oriented system. Based on a large industrial case study we evaluated the usefulness of historical data for the assessment of software architectures. In addition to the previously published technique, we show how *EvoLens* effectively supports the Relation Analysis process in terms of visualization, navigation, and analysis:

- *Revealing logical couplings*: Relation Analysis allows one to find change dependencies over time. Although no source code has to be parsed, the amount of processed data is still very large. This results from the fact that couplings are binary information relating single data items. Thus, the computational effort is $O(n^2)$. Despite this high number of findings *EvoLens* is able to provide visualizations very fast.

- *Architectural shortcomings*: Many architectural deficiencies can be discovered through metrics based on historical data. Examples of such findings are spaghetti code, blurred interfaces of components, and poorly designed inheritance hierarchies. These software anomalies are even easier to detect with *EvoLens*. We use color to indicate size and growth metrics of software components. Due to the coloring evolution anomalies can be spotted easily. The intensity of change dependencies between classes provides further hints at what kind of "bad smell" we recognize and where to start resolving it.

- *Frequent changes between system blocks*: Relation Analysis revealed many internal and external dependencies. Internal links are likely to point out limitations within packages. External couplings are even more challenging, because they may bring to light limitations of the architecture of the entire system. With the navigation capabilities of *EvoLens* the developers can explore the software structure as a whole and interactively reach any point in the software. Further, any point in time may be also reached by specifying the time window of particular interest.

- *Simple navigation*: Due to the huge base of results as output of the Relation Analysis, visualization and navigation of the information space improves findings of architectural hot spots. *EvoLens* uses only minimal

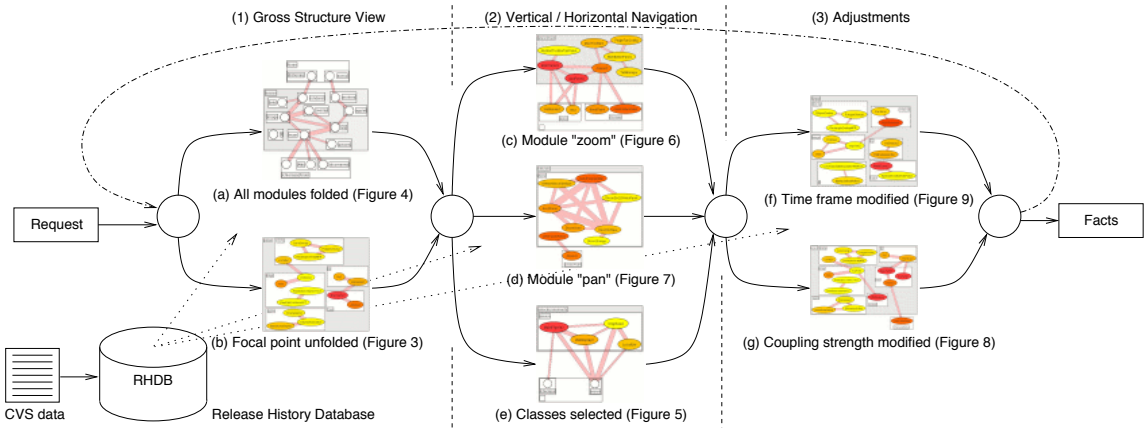

Figure 10: The *EvoLens* analysis process

information to fulfill its function. As a result *EvoLens* can be applied on very large legacy systems. Information about the historical development of software is often available, as it was demonstrated by using CVS.

5.1 Effective use of EvoLens

The features of *EvoLens* provide many possibilities to "walk through" a software system and its evolution. But it is important to guide an engineer in this analysis process so that we provide an effective use guide as follows:

Step (1) of Figure 10 is concerned with folding and unfolding of the gross structure of the software system under study. Folding reduces the scope from class level to module level: (a) depicts this state; whereas unfolding opens up modules and shows their class content in (b). It is important to note that all visualizations include the logical coupling information between any software parts. In the folding and unfolding, the engineer marks all interesting software parts: *EvoLens* allows an engineer to select arbitrary classes and/or modules by mixing the level of detail. Classes of some unfolded modules can be selected together with folded modules. As a result, *EvoLens* gets a user-defined set of software parts of particular interest.

Depending on the strength and intensity of the graphically shown logical couplings, in step (2) the engineer then navigates vertically or horizontally through the system hirarchy: s/he defines the new focal point either by clicking on a module: (c) represents navigation through zooming and (d) navigation through panning, or (e) by selecting an arbitrary number of classes. This defines the new major point of attention for the further visualizations. As a result, the lens view can be generated and shown to the engineer.

Step (3): For any time frame—(f) depicts this intermediate-step—the engineer might be interested in, *EvoLens* calculates the evolution data given the marked set of software parts and the focal point. Further, the engineer can define a logical coupling threshold in (g), above which the logical coupling edges are computed and visualized. This enables to consider situations of many common check-ins and fine-tune the graphical outcome generated by *EvoLens*.

The lens view then can be used to navigate in the software system providing specific lens magnifications for each collection consisting of: a set of observable software parts on module and/or class level, a focal point, a coupling threshold, and a time window.

Since each part of the collection can be adjusted by the engineer, the resulting evolution visualizations can be really effective. Of course, *EvoLens* offers so-called "convenience views" with pre-defined parameters for logical coupling thresholds and time windows for a faster analysis to the engineer. Lens view settings can be saved for later reuse in the tool together with the analyzed software system. Additionally, this allows us to save lens view settings that can or should be used for software systems within the same application domain. For the future, we are also thinking of exploiting this feature for a more systematic analysis of potential system families.

6. RELATED WORK

Several research projects have proposed utilization of visualization in software maintenance activities such as [12]. Our approach differs from previous works that it combines two dimensions of information (structure and evolution) in nested graph views. We enrich these representations with the description of additional metrics through color. With the help of folding the user receives a good overview of the interrelationships between modules. However, our emphasis is on the presentation of details in relation to other approaches such as [2]. For an in-depth information we refer the reader to one of the studies on visualization taxonomies [14, 17].

6.1 Evolution

Longitudinal empirical studies show potential in identifying phases in the life cycle of software where different activities need to be set to stabilize the development and maintenance process. A stable process is the foundation for a product of high quality. By focusing on the types of changes, costs and efforts to evolve, Kemerer and Slaughter [11] suggest that future trends within a particular system are predictable. Coupling and cohesion measures are a way to measure structural

cohesiveness of a design. The basic idea is that the more dependencies exist among modules, the less maintainable the system is because a change in one module will necessitate changes in dependent modules. Our measures may be used not only as coupling measures to guide restructuring efforts but also to validate the effectiveness of predictive and code-level coupling measures [10].

Clustering techniques may be used to improve modularity and support evolution. A measure for similarity or dissimilarity between two objects has to be identified to subdivide objects into clusters. Chung-Horng Lung [13] presents examples for utilizing clustering techniques for the improvement of software architecture. He suggests incorporating reverse engineering tools to identify the dependencies among classes. Zimmermann et al. developed a methodology for evolution mining based on CVS version archives [23]. They extracted rules from history data to enable a fine grained decision on which software items have to be considered too, if a programmer changes some part of the software. They evaluated their approach to find out how limited such predictions are.

The development process has large impact on program complexity and affects software related events such as release dates. Negative effects on software can be detected by examining logs of the source code repository [9]. Mathematical concepts from information theory can guide the investigation of software evolution. Textual descriptions of changes often indicate the type of the performed change. Mockus et al. discovered that a strong relationship exists between the type and size of a change and the time required to carry it out. This supports the necessity to visualize additional attributes of changes within the graph representation of software evolution.

6.2 Graph Visualization

Furnas specified general fisheye views. He explored a general formalism and implemented a first graphical program using this type of views. Fisheye views are an important representative of focus+context visualization. They account for the fact that humans often perceive their vicinity in detail and remote regions in successively less detail [6]. Fisheye views were applied on graph visualizations. Sarkar and Brown [19] contribute a general transformation to fisheye views to introduce layout considerations into the fisheye formalism. They interpret the size and level of graph items as functions of an object's distance from the focal point.

Storey et al. combined hierarchical views with focus+context and multiple perspectives into a single tool [20]. In our approach, we use graphical elements in the following way for user interaction: we use colors to depict metrics–dependent on the cumulated value for the selected time frame–such as complexity or coupling dependent on their; the width of edges is used to visualize a metric related to the linked nodes, e.g., strength of logical coupling; different types of source model entities are depicted via different node-shapes; within the different views, edges are always connected with lower level source model entities such as classes or sub-modules rather than their surrounding modules. This is in contrast to the SHriMP approach were edges frequently connected to higher level source model entities without revealing the exact connection endpoint. As outcome the folding and unfolding operations are perceived in a more intuitive way. In addition to the navigation within the static structure, i.e., class and module hierarchy, *EvoLens* supports the declaration of time-frames as selection criterion for dynamic data within the historical context, i.e., evolutionary coupling.

7. CONCLUSIONS AND FUTURE WORK

Visualizing evolutionary aspects of software evolution can help the engineer identify hot-spots of design erosion or structural decay rather quickly. Although many tools exist that provide zooming-in and -out within the hierarchical decomposition of a software system, only very few allow an engineer to view a system through a kind of lens view. We presented *EvoLens*, a visualization approach and tool for efficient explorations of evolution data across multiple dimensions. *EvoLens* is based on structural and temporal lens views, a technique similar to fisheye-views. But the graphical representation of *EvoLens* integrates enhanced zooming by navigating through software hierarchies with arbitrary selectable groups of software parts across module or package boundaries.

All historical and structural information is retrieved from CVS and stored in our Release History Database for uniform and fast access. From that we identify all logical couplings that represent many common modifications of particular software parts over a significant number of releases. This evolutionary coupling is then visualized in the structural and temporal lens views and available for navigation and user-defined analysis.

As a result *EvoLens* allows an engineer to define a focal point for the lens view and navigate along the time dimension by user-defined time windows. The comprehension is supported by using color for metrics of the software modules or classes. *EvoLens* facilitates this analysis process of a software system's structural evolution through the following navigation functions: (a) lens views for intuitive orientation; (b) vertical "zooming" and horizontal "panning" functions to navigate through the structural information space; (c) folding and unfolding of structural entities such as modules; and (d) specification of time frames and selection of entities on quantitative basis.

In contrast to other approaches we rigorously use *projection* and *focus+context* in our visualizations. The *EvoLens* prototype tool has been evaluated on basis of a medium-sized Java application consisting of 580.000 LOC over an 18 months evolution period. The navigation functions together with our visualization approach has shown to be an effective and intuitive way to quickly locate and highlight areas of evolutionary coupling in nested structures of large software systems.

In future work we will enhance the layout of the graphs: we will further investigate the placement of the depicted items. At the moment we use a force-directed method to handle the layout [5]. Additionally, *EvoLens* provides abilities to manually correct the layout, but this should be better supported. As this prototype is an Eclipse plugin, we plan to make it available as open source to a broader community.

8. ACKNOWLEDGMENTS

We thank our industrial partner that provided the case study and helped us with the interpretation of the results. The work described in this paper was supported by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT), The Austrian Industrial Research Promotion Fund (FFF), and the European Commission in terms of the EUREKA 2023/ITEA project FAMILIES (`http://www.infosys.tuwien.ac.at/Cafe/`). We further thank Andrej Sramko (Vienna University of Technology, email: `sramko@infosys.tuwien.ac.at`) who implemented the Java prototype of *EvoLens*.

9. REFERENCES

[1] P. Cederqvist. *Version Management with CVS*. Network Theory Ltd., December 2002.

[2] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. *Proc. of the 2003 ACM symposium on Software Visualization*, pages 77–86, June 2003.

[3] J. Dill, L. Bartram, A. Ho, and F. Henigman. A continuously variable zoom for navigating large hierarchical networks. *Proc. of the 1994 IEEE Conference on Systems, Man and Cybernetics*, pages 386–390, October 1994.

[4] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings International Conference on Software Maintenance (ICSM'03)*, pages 23–32, September 2003.

[5] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.

[6] G. W. Furnas. Generalized fisheye views. *Proc. CHI '86 Conf. on Human Factors in Computing Systems*, pages 16–23, September 2003.

[7] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance*, pages 190–198. IEEE Computer Society Press, March 1998.

[8] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. *Proc. of the 6th International Workshop on Principles of Software Evolution*, pages 13–23, September 2003.

[9] A. E. Hassan and R. C. Holt. The chaos of software development. *Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, page 84, September 2003.

[10] M. Jazayeri. On architectural stability and evolution. *7th Int. Conf. on Reliable Software Technologies - Ada-Europe 2002*, June 2002.

[11] C. F. Kemerer and S. A. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, July-August 1999.

[12] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. *LMO 2002 Proceedings (Languages et Modeles a Objets)*, pages 135–149, 2002.

[13] C.-H. Lung. Software architecture recovery and restructuring through clustering techniques. *Proc. of the 3rd Int Workshop on Software Architecture*, pages 101–104, November 1998.

[14] B. A. Myers. Taxonomies of visual programming an program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[15] D. L. Parnas. Software aging. *Proc. of the International Conference on Software Engineering (ICSE 16)*, pages 279–287, May 1994.

[16] T. M. Pigoski. *Practical Software Maintenance*. John Wiley and Sons, New York, 1997.

[17] B. A. Price, I. S. Small, and R. M. Baecker. A taxonomy of software visualization. *Proc. 25th Hawaii Int. Conf. System Sciences*, 1992.

[18] V. T. Rajlich and K. H. Bennet. A staged model for the software life cycle. *IEEE Computer*, 33(7):66–71, July 2000.

[19] M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 83–91. ACM Press, 1992.

[20] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Mueller. On integrating visualization techniques for effective software exploration. In *Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, pages 38–45. IEEE Computer Society, 1997.

[21] A. von Mayrhauser and M. A. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, pages 44–55, August 1995.

[22] T. Zimmermann and P. Weibgerber. Preprocessing cvs data for fine-grained analysis. In *In Proceedings of International Workshop on Mining Software Repositories (MSR'04)*, May 2004.

[23] T. Zimmermann, P. Weigerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *Proc. of the 26th Int. Conf. on Software Engineering (ICSE'04) - Volume 00*, pages 563–572, May 2004.