Università
della
Svizzera
italiana

Faculty
of Informatics

METROX Framework

A Standalone Framework to Visualize Software Evolution

Bachelor Project

Paolo Domenighetti

Supervised in 2009 by
Prof. Michele Lanza

Abstract

Software visualization is the visual representation of information gathered from software applications about their architecture, design, behavior and evolution. In our case, software visualization is a powerful tool to represent evolving software stocked in a versioning repository by using current software metrics. Since versioning repositories can always be queried to obtain a working copy for a given revision and since revisions are directly tied to time, working copies can be considered snapshots taken on the history of a project or software. It is therefore natural to apply metrics computation on software entities existing in a given working copy to construct a history of evolving metrics values. Moreover we believe that aggregating different types of metric measurements performed at multiple level, from revisioning activity itself to design metrics, will support and enhance sofwtare evolution analysis to easily find out past causes of current problems and to forecast possible issues in the future life of a software application.

Contents

1 Introduction **1**

2 Approaches to Software Evolution **2**
2.1 Choosing Metrics for Software Evolution 2
2.1.1 Revisioning activity 3
2.1.2 LOC - Lines of Code 4
2.1.3 Design metrics 4
2.2 Use of polymetric views and the Evolution Matrix 4
2.3 Aggregating Different Types of Metrics 5

3 The METROX Framework **7**
3.1 Data, Metrics and Transformations 7
3.2 Gathering and handling information 10
3.3 Visualizing evolution information 10
3.3.1 Statistics perspective 10
3.3.2 Evolution perspective 11
3.4 Framework Architecture and Implementation Choices 13
3.4.1 Architecture and Components 14
3.5 Discussion 15
3.5.1 Scalability 15
3.5.2 Usability 15
3.5.3 Drawbacks and Problems 16

4 Validation and Testing **17**
4.1 Case Study A: Medium Size Repository 17
4.1.1 Performance 17
4.1.2 Validity 17

5 Conclusion **21**

List of Figures

2.1 An example of a software metrics visualization [4] 3
2.2 Author's commit activity visualized in graph chart (http://www.statsvn.org/ (c) StatSVN 2009) 3
2.3 An example of a polymetrics visualization tool [6] 5

3.1 A screenshot of the METROX Frameworkin the statistics perspective. . . . 8
3.2 A screenshot of the METROX Frameworkin the evolution perspective. A separated window has been used to render the real time 3d environment. 8
3.3 The 7-tuple graphically represented 9
3.4 The statistics perspective with the Selected JavaSource (total) Activity plot for a selected source. 11
3.5 The 3d environment window in detail. It is possible to note how sizes and colors work together in a polymetric view. 12
3.6 Schematics showing the overall architecture of the framework 14
3.7 The very simple user interface of the Evolution perspective with the Tree package list on the elft and the time navigation slider right on top. 16

4.1 A view of the entire SVNKit project's activity. 18
4.2 A view of the entire SVNKit project. Time dimension goes toward the observer. 18
4.3 A look at a very active entity in package in EVOLUTON 2 view `org.tmatesoft.svn.core.intern` revisioned almost at every snapshot taken on the repository. Notice sourrounding entities belonging to other packages that where committed finished at the beginning of the project and that has no more been modified. 20

List of Tables

3.1 Visualizations and related metrics as instances of a 7-tuple 13

4.1 Framework performance analysing SVNKit's repository with a time granularity of 7 days (i.e. a time unit is 7 days and snapshots are taken at each time unit) 19
4.2 Framework performance analysing SVNKit's repository with a time granularity of 21 days (i.e. a time unit is 21 days and snapshots are taken at each time unit) 19
4.3 Repository information gathered in the first pass 19
4.4 Snapshots analysis using X-Ray [http://atelier.inf.unisi.ch/ malnatij/] as a verification and validation tool 19

Chapter 1

Introduction

Software versioning systems are today common in software industry which uses them extensively. The reason behind this choice is that versioning systems are suited for efficient sharing of projects and, as their name suggests, they allow analysis of the projects changes and their evolution, keeping information for the entire project's life. Tracking of expansions or contractions of a project is therefore easier and change trees created along the project's life are a valuable set of data to perform historical analysis and forecast of the future software evolution.

Versioning systems, among them CVS and SVN Subversion, provide - or can be equipped with - a huge number of client software providing good statistical analysis and repository inspection. However most of them are based on standalone tools and only few of them use visualisation to present the extracted information. In general, such evolutive visualizations are tied to a given scope and level of abstraction or domain: some of them produce visualizations using versioning activity data (mostly focused on developer's activity or other repository's stocked infomration) whereas others focuses entirely on language paradigm and provide object-oriented metrics visualizations focused on design changes and complexity. Hence the main purpose of these tools is to furnish analysis in one particular level of abstraction as well as to try to inspect and highlight problems in a given level of abstraction by gathering and presenting information uniquely tied to that level of abstraction. But problems in a software's evolution may be caused by lacks and errors repeated at many levels: so, for example, a given entity bug in an entity can very well be caused by an erroneous management of the activity and coordination among other sources and by a complex design structure simoultanously.

Since software do change in time at many level, seeking common causes to constant or redundant problems in design and quality need a multiple views approach, where simultaneous comparison of different sofwtare metrics are aggregated to present a mixed domain visualization. We believe that, putting all this information together and aggregating different types of metrics (especially versioning metrics and design metrics) to produce software visualizations, will help the user to see where problems or difficulties occured during the development of a project and most of all he will possibly be able to assess and forecast possible future problems in the software's incumbent evolution.

Chapter 2

Approaches to Software Evolution

Versioning systems provide a huge amount of metadata. Generic versioning systems - such as CVS or Subversion - are suited for a variety of different programming languages. Actually the meta data they provides targets files and directories' versioning, revisioning and logging. Although versioning systems do not provide analysis directly, versioning history can be mined and analyzed using fast algorithms.

To cope with such a huge amount of data, and thus structural software complexity, requires techniques that can resume complexity (without losing detail) in an intuitive way, understandable by the human cognitive capabilities. One of this techniques is software visualization: the visual representation of information gathered from software applications about their architecture, design, behavior and evolution. In our case, software visualization is a powerful tool to represent evolving software stocked in a versioning repository by using current software metrics. Since versioning repositories can always be queried to obtain a working copy for a given revision and since revisions are directly tied to time, working copies can be considered snapshots taken on the history of a project or software. It is therefore natural to apply metrics computation on software entities existing in a given working copy to construct a history of evolving metrics values.

2.1 Choosing Metrics for Software Evolution

Large amounts of data coming from repositories can be classed in two different types: sources themselves (which compose the working copy) and metadata gathered by the versioning systems (e.g. log entries submitted by developers when comitting changes on one or more sources). However results obtained from mining versioning system are not sufficient because they are common to all sorts of projects and to different programming languages' paradigms. Thus working copies' sources generated by repositories have to be recognized as belonging to a given programming language and paradigm. Sources can then be inspected and the encapsulated information can be interpreted as representing an entity to be measured.

Going from the highest level of generalization (repositories themselves) to the lowest (entities belonging to a system developed in a given programming language) different classes of metrics with different targets are needed.

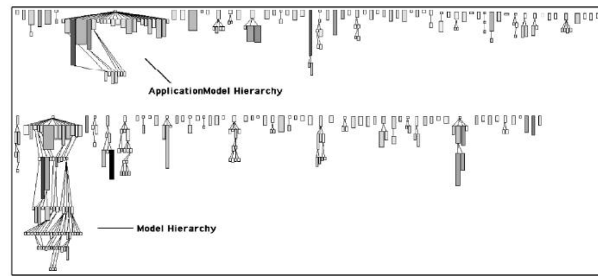

Figure 2.1: An example of a software metrics visualization [4]

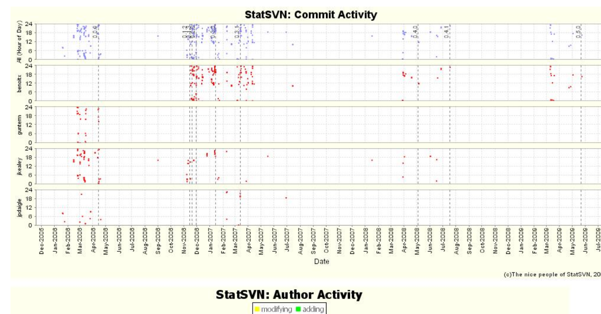

Figure 2.2: Author's commit activity visualized in graph chart (http://www.statsvn.org/ (c) StatSVN 2009)

2.1.1 Revisioning activity

Revisioning activity, i.e. commit activity produced by developers, is not only a way to observe and control developer's work. If commit activity is targeted on sources, instead on developer's, then it is possible to analyse the source changes' ratio and observe its birth, growth and eventually its dead. When relationed to other sources' life evoltuion, the entire project can be seen as living organism whose organs (components) change and evolve through time. Actually a source does not only change in size but can obviously change in structure keeping its size constant through time. Hence different types of changes may imply different intensities in revisioning it: for example, a very active source whose size do not change (e.g. lines of code) is expected to show symptomatic problems where complex methods seem to be hard to debug or the entire source (entity) has many bugs and/or incoeherencies, thus the high activity ratio performed to debug it.

Building and providing a revisioning activity does not mean provide a complete history of the evolution of a system. Version control tools do not store all the information generated by developers. They do not record every intermediate version of every single source, but only snapshots taken when a developer explicitly commits source code into the repository. So far, the activity ratio shown by commit actions can be far from the reality: a high ratio activity source could have been modified even more as what it appears thus develoepres could have had even more problems to face when debugging/testing the source. However, unless using specific methods [5], commit activity can be a genuine source of information and the most objective one, because it was registered in the repository.

2.1.2 LOC - Lines of Code

Descending to a more specific level, lines of code (or LOC) is possibly the most widely used metric to measure a software's size. There is a number of definitions for the number of lines of code in a particular software. The most common definition of LOC counts lines that are not blank lines or comment lines in the source code. LOC count can of course be applied to single entities defined in single sources, like classes or methods defined in JavaSources in our case. We observe polyfunctionality of LOC because it can be used for any source file or it can be focused to an interpreted entity or subentity of a system.

It is also possible to push LOC analyisis to the activity level we discussed previously. Pairing commit activity with LOC ratio changes between revisions increases considerably the ability to understand changes within a system and which could be the cause of these changes. High activities and high LOC ratio may, for example, imply that the source has not yet acquired a defined structure and hence the proper functionality it is supposed to have has not been well planned at the design stage.

2.1.3 Design metrics

The main subject whose instances have been analysed through this bachelor project are Java repositories, i.e. repositories where the majority of sources are Java Sources. Design analysis and evaluation are therefore necessary to achieve comprehension of the software evolution. The first point to consider is that quality of a design is strongly influenced by the systems package relationships. So a good place to begin assessing design evolution is by evaluating package relationships changes. Measuring a systems level of reusability and maintainability (through packages and imports analysis) can serve as a quality feedback about the structure of a software.

Other useful design metrics that may better highlight the complexity of an entity in relationship with the structure of a system are the well known object-oriented design metrics, whose computation is directly achieved through the scanning of the source code of an entity. Measurements are direct in the sense that they do not need other variables or metric values to be computed. Thus these metrics are good basic elements to use as parts of the metrics aggregation technique used and tested with METROX Framework. Among them, the Method's Lines of Code (LOC) and the number of methods in a given entity (NOM).

2.2 Use of polymetric views and the Evolution Matrix

Polymetric views are a powerful visualization techniques that uses graphical representation to present multiple metrics relationships existing in a given point of the software evolution. The purpose underlying to this methodology is to visualize the software structure through its metrics captured at a given moment in time using geometrical objects whose dimensions (sizes) are computed . A polymetric view is therefore defined and issued placing geometrical objects in a reference system (e.g. a three dimensionsl vectorial space) where instrinsic object's properties such as position related to origin and dimensional sizes can be used to represent metric values. Obviously, since metrics values belong to different numerical sets and interval, the plymetric view by defition has to normalize them to keep the visualization consisten and intuitively tied to a given region in space. This can be done, for example, by simply normalizing metrics using metric units (that can be avarages of the metric series collected) and placing them in a limited geometrical region (e.g. the bounding box used in [6])

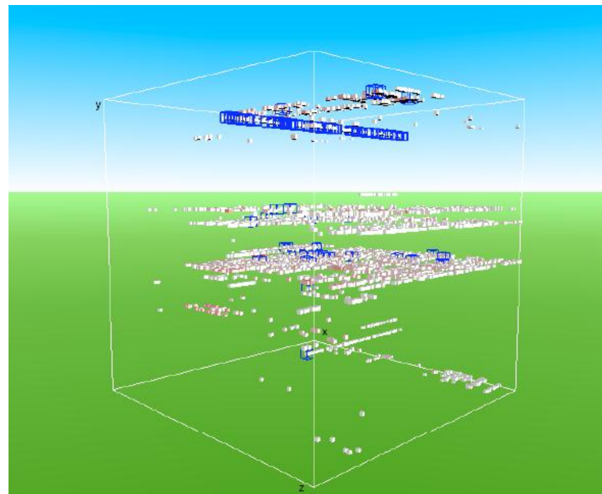

Figure 2.3: An example of a polymetrics visualization tool [6]

Another powerful tool, defined and discussed in [2], is the Evolution Matrix. The evolution matrix displays the evolution of the classes of a software system. Each column in the matrix represents a version of the software, while each row represents the different versions of the same entity. Elements (entities at a given temporal position) of the evolution metrics are represented using a polymetric approach where dimensions of the shapes used to represent an entity are metrics previously chosen. By looking at the rows it is then possible to intuitively recognize entities' metrics evolution and to devise some sort of behavior followed by the entity through time.

Fusioning this two visualization approach we can issue a three dimensional representation of a set of adapted three dimensional evolution matrixes placed together in a spatial reference system.

2.3 Aggregating Different Types of Metrics

In our opinion polymetric views' power and ability to visualize software is tied to the kind of correlation between metrics that they convey. In general, we observed that many polymetric visualization tools produce representation of a single category of metrics. For example, common polymetric views represents entities subject to some versioning activity and build relationship between metrics in that domain (e.g. [6]). In the same way, other views put object-oriented design metrics together to achieve a design compelxity isualization.

We believe that polymetric three dimensional views can be pushed even further in their role, making them represent multiple metrics belonging to various different categories and that placing them in a evolution history will increase the ability of a software evolution researcher to assess relationship between the design complexity of a system and the revisioning activity producing it. To realize this we will:

Build a software application that performs polymetric visualization using metrics belonging to different categories: versioning, system size and design metrics;

Test the software on medium and large size reposiotories to be able to acquire huge datasets, produce three dimensional polymetric visualizations embedding evolution matrixes in a three dimensional environment;

and finally highlight the possible relationships between different kinds of metrics (if they exists) and discuss further investigations.

Chapter 3

The METROX Framework

In this chapter we define metrics used to solve our problem. Moreover we discuss how data is gathered from repositories and evolving sources, what kind of measures and analysis we perform on them and finally which metrics we can build from acquired information and how we use them to visualize software evolution. We also present a tool that could be considered an instance of a possble solution to the problem mentioned in the previous chapter: the METROX Frameworkapplication. We will also present its overall architecture and the visualization features it provides. Then, in the next chapter, we present some tests performed for validation purposes over a set of replicated projects taken from Sourceforge.net.

3.1 Data, Metrics and Transformations

The METROX Frameworkis an application to visualize software evolution whose core goal is to purpose a 3d visualization environment. In the previous chapter (Chapter 2) some of the most important and frequently used software metrics have been defined. Since the purpose of this project is to try to provide a possible solution aggregating different types of metrics, it is imperative to define what metrics we used in our application and how they have been transformed to better solve a basically practical problem: how do we represent them graphically.

Visualizations in a 3d space have to be geoemtrically coeherent and, at the same time, avoid confusion when graphically rendered. Thus our space will first of all as a 3 dimensional vector space whith an origin (the null vector) and three base vectors that build up the three x, y and z-axis. Sinc visualizations are actually time histories, one of this axis is used as a time unit axis. Therefore positioning of objects in this space cannot be used to represent metrics because different values would translate solids representing entities everywhere in the space and thus the visualization will lose clarity. Metrics are instead represented with sizes (sides lengths of the solids for the 3 dimensions) and color. A visualization in this space will then be defined as 7-tuple $(x, y, z, sx, sy, sz, col)$ whose terms are groupings (for positions) and the different kind of metrics' transformations (sizes and color).

We decided to name and define metrics transforations not because nobody did this before us but because to solve our practical problem these definitions are specifically suited for our visualizations. As long as it is possible original metrics discussed in Chapter 2 will be named in the same way and will of course have the same definition. The following list is not complete and a huge number of other metrics (and transformation) can be created if it is necessary. We list only those that are implemented in

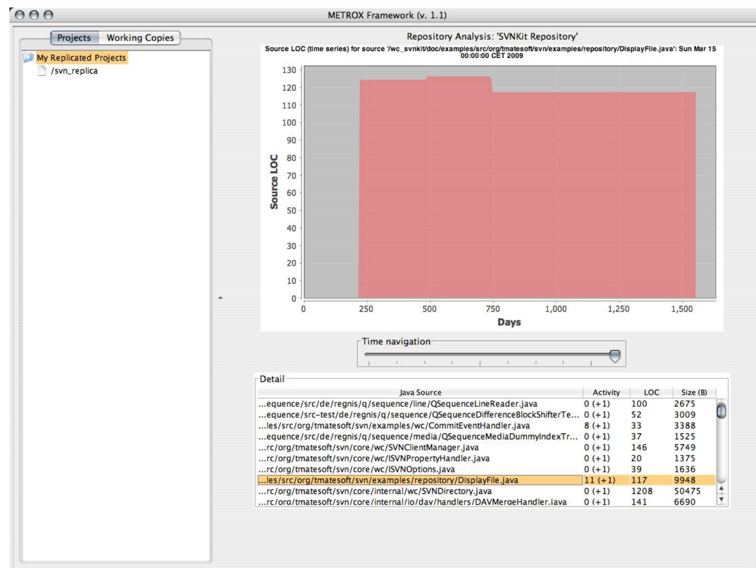

Figure 3.1: A screenshot of the METROX Frameworkin the statistics perspective.

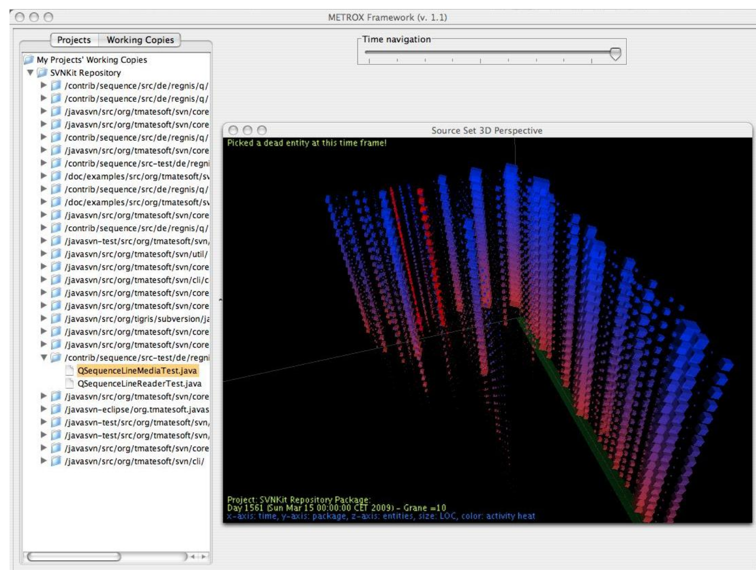

Figure 3.2: A screenshot of the METROX Frameworkin the evolution perspective. A separated window has been used to render the real time 3d environment.

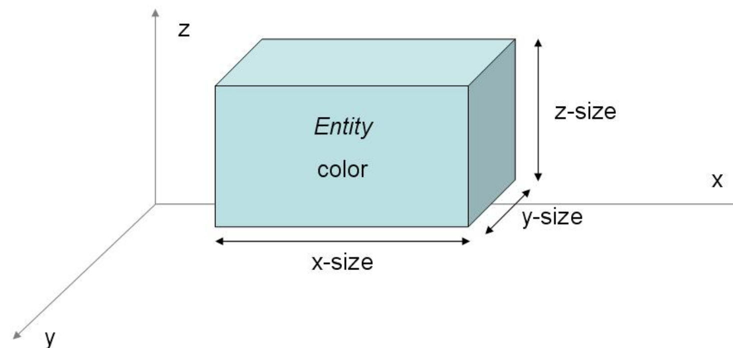

Figure 3.3: The 7-tuple graphically represented

METROX Frameworkand keep speculation about newer transformations in Section 3.3.

NLOC This fundamental transformation is the WLOC size of a given entity divided by the avarage of all entities' WLOC sizes at agiven time unit. This definition should not be taken as having an intrinsic mathematical meaning but it is rather a way to normalize WLOC to be fitted in a 3d reference system which has limited geometrical size. The avarage is a well know sensitive variable subject to variations when extreme values are present in its computation. However for our purposes, even when such extreme WLOC values appear for some entities, proprortion is mainaned in the visualization and comparation between entities is still possible because all other NLOCs are normalized with the same avarage. These remarks obviously apply to other trasformations.

DLOC The difference of WLOC of an entity between two consecutives time units, normalized with the avarage of all WLOC differences for the same consecutives time units.

RACT The ratio of activity performed on the entity. This transformation is almost always used as an instance of col. RACT is the only transformation that, for a given entity, can change whitout being affected by other entity's modifications. RACT can be compared to pheromone in an ant system (although its objective is not the same) or to materials heat: when activity on an entity is carried out, the entity will become hotter (and therfore its color will shift toward red) by a factor depending on the amount of activity itself and the speed activity increases (activity difference divided by time). On the contrary if no activity is performed and the entity is unchanged it will become colder (and therefore its color will shift toward blue) by a factor depending on the time units lived without changes.

NNOM Is the normalized number of methods, i.e. the number of methods of an entity divided by the avarage number of methods of all entities.

NLPM Is the avarge lines of code per method, normalized with the avarage of all lines of code per method of all entities.

IIR The inner imports ratio computed taking the number of inner imports of a give entity divided by the total imports of all entities. This is not a normalized value since imports are considered uniques.

3.2 Gathering and handling information

Using the Imports wizard, repositories are replicated by the application into the local disk: gathering information remotely would have been too slow thus this solution is the one that offers better performances. Replicating repositories can be time consuming but once it is done the local replicated drepository will contain all the original information (as well as all the versioning branches and log entries sets). Once the repository has been replicated, the framework initializes its analyisis process.

We distinguish two types of data gathered from projects' repositories: versioning information, as well as metadata acquired directly by the repository, and statistical data acquired in a second phase analysing the revision history of all possible working copies. The first type of data is acquired directly from log entries of a specific repository. We considered a preprocessing step followed by a caching process since it is impossible to perform retrieval of log entries on-the-fly: an active and popular project has approximatively tens of thousands revisions with hundreds of files, thus the speed and the responsiveness of the framework would have been deeply decreased.

Since the fundamental purpose of our solution is to provide an evolution history, the log entries themselves are obviously not sufficient to perform measures on evolving Java Sources: snapshots (i.e. working copies at a given revision - at a given time unit) of living sources have to be taken at each time unit to allow a source and design analysis. What follows is an sample set of data gathered from repositories and selected working copies during the preprocessing phase:

- **From a repository's log entries set:** developers, number and type of commits, changed paths (i.e. paths of changed sources), current revision, etc.
- **From repository's existing metadata:** date of creation, number of revisions, log entries set size, overall size, authorities, versioning branches etc.
- **From source files (existing in a given snapshot):** lines of code, abstract source model, package depth, inner and external imports, etc.

3.3 Visualizing evolution information

Our framework offers two main visualization perspectives: a statistical visualization perspective which uses charts to show the evolution of some useful properties of a replicated project; and an evolution perspective that offers multiple kinds of 3 dimensional views whose key role is visualize the software's evolution. Perspectives can be changed by switching tabs on the left.

3.3.1 Statistics perspective

The navigation tabbed panel lists replicated projects (repositories) imported by the user. Selecting an item of the list will make the perspective switch to the given repository context and all visualization features will be relative to the selected item. Then, navigating through the menu, visualization types can be selected. What follows is a list of visualizations featured by this pespective. They aggregate charts with general and specific descriptions of the measures and metrics relative to the selected project. A list with descriptions of the charts views is given below.

- **General Repository Activity** A histogram that shows the general repository activity classed by type. Types are the types of actions performed by developers when

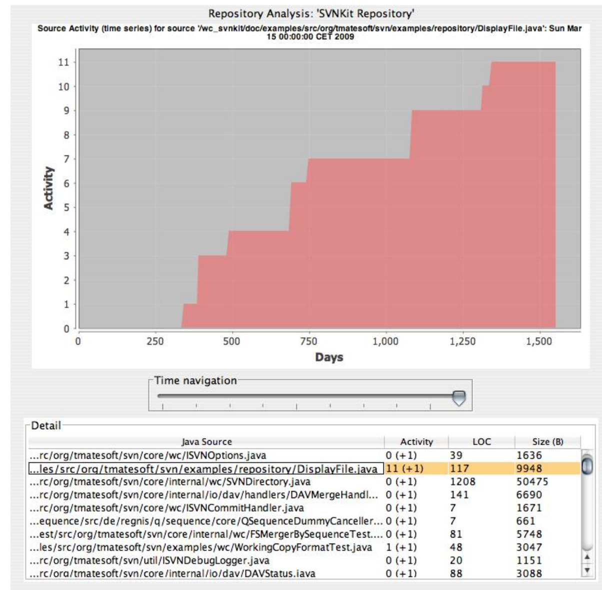

Figure 3.4: The statistics perspective with the Selected JavaSource (total) Activity plot for a selected source.

performing their commits: Additions, Deletions, Changes (Modifications) and Removals. Data visualized are cumulative but they are obviously not graphed in this way, because of the nature of the histogram chart itself.

- **General Activity Plot** A plot chart that represents repository's activity type-mixed. This view does not allow recognition of different kind of activities but cumulates over time activity at each time unity.
- **File Count Plot** A plot chart that shows the total file count of the repository. This includes every type of file (including property files and pictures for example).
- **Cumulated Activity Plot** Is similar to the general activity plot but shows a multiple plot chart recognizing the four different kinds of activity.
- **Total LOC Plot** A plot chart graphing the sum lines of code of all sources.
- **Selected JavaSource Activity Plot**
- **Selected JavaSource LOC Plot**
- **Selected Developer's Activity Plot**
- **Selected Developer's LOC Plot**

Although some measures or metrics are relative to the entire repository (focusing source files and many other types of files) it is possible to acquire information from single files and see their evolution one by one. This visualization modality is applied to developers too: the user can select single developers to see their activity and their production of lines of code.

3.3.2 Evolution perspective

The purpose of a 3d metric visualization is not to display as much information as possible but to offer views whose combination of metrics can show useful information to the

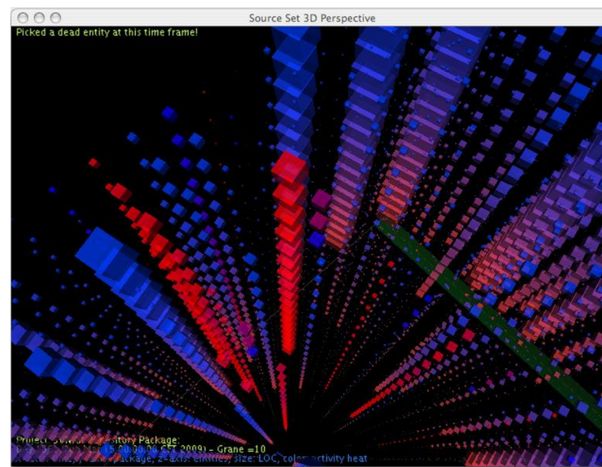

Figure 3.5: The 3d environment window in detail. It is possible to note how sizes and colors work together in a polymetric view.

user. Volumes used are therefore simple and special 3d graphics texturing and lighting are obviously not necessary. However the use of a 3d space implicitly allows a greater number of information to be displayed, compared to a bi-dimensional visualization. In this perspective the user can navigate through entities and compare them to each other. Position coordinates in a 3 dimensional reference system let us place objects using 3 classification (or grouping). The size span (i.e. heigth, width and depth of solids) over the 3 dimensions can be used to represent a maxium of 3 different measures, or metrics, such that it is possible to make a direct correlation between them. Finally two more dimensions - although they are not spatial dimensions in strict sense - can be used to represent measures and metrics: colors and transparency. Thus a solid in our 3 dimensional visualization is generalized to a 7-tuple $(x, y, z, sx, sy, sz, col)$ as we have already pointed out in Section 3.1.

METROX Frameworkfeatures 3 classes of visualization distributed over five specific instances of the 7-tuple discussed previously. Each type can be more or less be considered a practical attempt to provide a solution to the problems stated in Chapter 2. As we can see from (fig. 3.2) the Evolution perspective allows the user to point every time unit in the evolution history through a slider. The user can also calibrate the time resolution of the represented history by using the 'g' or 'f' keys. In visualizations where one of the axis is the time axis this feature change the period interval between snapshots, enlarging or restricting it. This is useful when, focusing the end parts of the history, the visualization become too much populated.

What follows is a description of each visualization used and how they are rendered. Each of them is represented showing three reference axis (for the 3 dimensions, red green and blue for x, y and z axis respectively) and using a base bounding rectangle to focus the selected package (when in multiple package visualizations). Moreover each description provides a motivation behind the choice of a given metrics aggregation.

EVOLUTION 1 Athough every visualization focuses on a project's evolution EVOLUTION 1's purpose is to represent sources size (normalized WLOC) and imports evolutions coupled with its RACT history (for a single selected package). This visualization represent the first attempt to couple versioning metrics with size metrics without focusing on design.

Name	x	y	z	xs	ys	zs	col
EVOLUTION 1	Class/Int.	time	N/A	IIR	NLOC	NLOC	RACT
EVOLUTION 2	Class/Int.	time	pkg	NLOC	NLOC	NLOC	RACT
VERSIONING	Class/Int.	time	pkg	NLOC	NLOC	NLOC	DLOC
DESIGN 1	Class/Int.	time	pkg	NLOC	NLOC	NNOM	RACT
DESIGN 2	Class/Int.	time	pkg	NLOC	NLOC	NLPM	RACT
LOC and Imports	N/A	N/A	N/A	NLOC	NLOC	NLOC	IIR

Table 3.1: Visualizations and related metrics as instances of a 7-tuple

EVOLUTION 2 A visualization similar to the previous one but focusing on all packages of a given project. Since the visualization space could become over crowded, we decided to avoid IIR and to represent entities as cubes whose side length represent normalized WLOC size (NLOC). RACT is represented with color heat (see pr.).

VERSIONING This visualization focuses on DLOC (i.e. LOC differences in entities between time units) in the evolution. To avoid overcrowded space and information, in this case too entities are solid cubes whose side length represent NLOC. The motivation of this visualization is to highlight possible correlations between RACT and NLOC, although this does not always happen (see Chapter 4).

DESIGN 1 This visualization is an attempt to aggregate the number of methods of a class, its NLOC size and the RACT over that class. Its purpose is to show possible correlation between NNOM and RACT (using *col* as previously detailed) in a time interval. Hign NNOM, paired to a high NLOC, in our opinion, should imply a greater and constant RACT along time.

DESIGN 2 Similar to DESIGN 1 it represents an attempt to aggregate NLPM, instead of NNOM, with NLOC and RACT. The motivation behind this choice is that, in our opinion, NLPM is a more realistic metric than NNOM to show possible correlation between revisioning activity and the method load of a class. It is expected that a class with many methods and also a high NLPM is more bug prone and therefore its RACT higher and constant along time (because of debugging).

LOC and Imports This visualization does not properly use dimensions as we discussed previously. It should possibly be taken as a ”general” coharse view of a project at a given time unit in the evolution history. Size of the cubes represent normilized LOC and the color an inner imports ratio metric (IIR). The motivation of this type is to give a fast view of the different entities and their LOC size.

3.4 Framework Architecture and Implementation Choices

The framework is a standalone application that does not require any external library, except OpenGL natives for Mac OS X. We first considered developing METROX Frameworkas an Eclipse plugin, but this would have strongly compromised the ability to perform real time graphic computations and to embed a 3d vizualisation environment into an Eclipse perspective itself. Therefore the framework has been converted to a standalone application which aloud us to use conveniently 3d graphics through the Swing GUI Library.

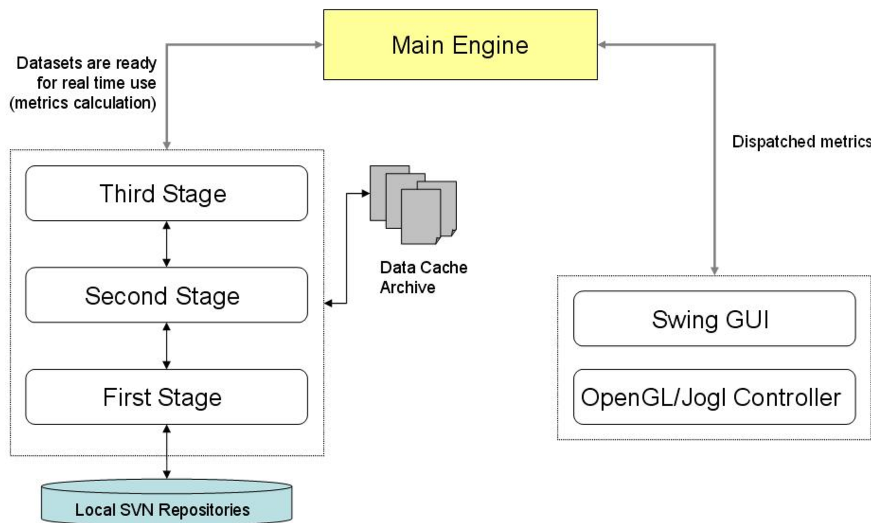

Figure 3.6: Schematics showing the overall architecture of the framework

3.4.1 Architecture and Components

The application is a composition of three core level components, a master component and one single User Interface layer. Core components can be considered three different stages of computations by the way they operate with or act on input data. Such components queries local repositories continuosly to accomplish the tasks described below using the SVNKit AP Interface (http://svnkit.com/). The Master component handles data and computations through the three core components and, after having performed statistics acquisition, it manages real time visualization holding cached application data.

Bottom stage This core component provides remote repository connection to perform Imports selected by the user, performs Subversion access and replication over remote repositories (commands `svnsync sync`) and queries local repositories to extract log entries relative to the given projects (commands `svn log [PATH]`). This process is carried out in one pass only for projects newly replicated and no other components are active in this phase. Existing replicated repositories are instead recognized and data acquired from cache files by the Middle stage.

Middle stage This component gather log entries concerning all the files contained in the repository branching as well as repository's main information themselves (i.e. revision numbers, directory trees, etc.). Data extracted from log entries allows the component to build the time unit set indexing and to build meausres listed previously in Section 3.1. Measured produced at this stage are already ready to be used by the application engine. However all of them are used to enhance analysis speed at the last stage of the data anlysis. Existing replicated reposiotires do not need the same inspection: data is cached once (as in the bottom stage) for reuse.

Top stage The third core component is responsible for the fine graned job. Analyisis of the repository is now focused exclusively on the revision history of Java Sources and a "snapshot" workingcopy of every source is taken at each time unit via the `svn update [PATH]` command. Data acquired at this stage is used both by the Statistics perspective and the evolution perspective. The master component uses specific sources' measures

acquired at this point to produce 3d visualizations. Existing replicated repositories (and replicated working copies) that where already analyzed once are recognized and do not need inspection.

Master component The application engine holds runtime data for current active repositories. Its main task is to dispatch history data to the perspectives that need them. The main engine also controls data filtering and time resolution as well as the cumulative computation of measures and metrics over a time period. Moreover it practically provides series of snapshots of a project history over time span decided by the user using the History Navigation sliders described previously.

UI and Graphics component Graphics component is marginal for the purposes of this text. However without the 3d graphical environment aloud by the Evolution Perspective the framework would look quite different. The graphical engine of the framework is very simple and is built directly on top of the Jogl API(footnote)

3.5 Discussion

There are many issues - either theoretical or practical, at this point, that can be discussed. Further investigations on the validity of the METROX Framework, its performance and the question of whether or not it can be useful to solve problems questioned are taken in the next Chapter. What we investigate here is the acceptability of the metrics and their use. Furthermore we discuss technical issues about the system and how the framework itself could be improved.

3.5.1 Scalability

METROX Frameworkis open to further expansion. Visualizations offered in both perspectives can be considered modules that can be detached and reattached to the system. It is possible to extend firsts' stages analyis to pick new metrcis (e.g. new design metrics) and to pack them in a new module. Thus even new modules can be added and new visualization's 7-tuples can be appended to the framework whithout making anychange to the system. However, since performing measures and picking metrics is time consuming, too many modules could decrease the overall performance of the system. It has to be said that what we mean here is that new visualizations can be created by implementing them. The framework does not allow for visualization's customization (See below).

3.5.2 Usability

We distinguish two issues regarding usability: the framework overall usability and the visualizations usability. For the former, since the main purpose of the framework is to provide visualizations (possibly intutiive graphical representations of a software evolution), visualizations comes already set in the most complete way: the user can change the object of each visualization by simply navigating in the Tree on the left. Time navigation is also easy to perform via a 'movie' slider. Visualizations are easily changed by swithcing them in the Evolution menu and graphics will render the evoluton of the focused project and/or package.

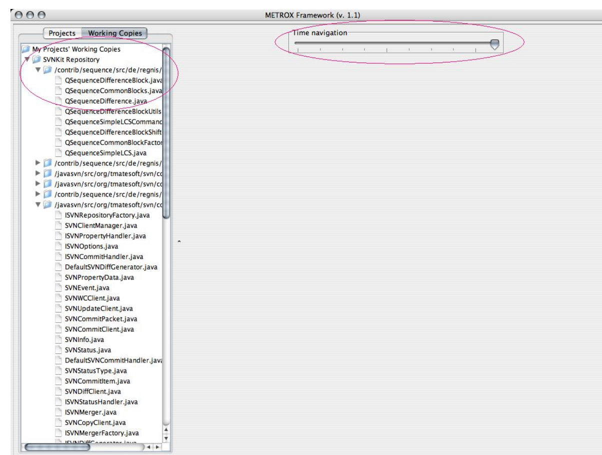

Figure 3.7: The very simple user interface of the Evolution perspective with the Tree package list on the elft and the time navigation slider right on top.

3.5.3 Drawbacks and Problems

The framework has to cope with huge datasets built when scanning and analysing the repositories and the selected workingcopies. Once this information has been extracted it is cached in a file (for further initializations) and also kept in the runtime memory of the application. This enhance performance when retriving data to produce visualuzations. However, when the projects ecosystem size loaded by the application is too high, runtime memory is fast filled and responsiveness drop.This problem may be partially solved by not loading all information for every repository/project once at the beginning and doing it just in time through a database.

Another great lack of the framework is the ability to customize visualizations in the Evolution perspective. However this can be created later in a new version: the framework has to be considered a tool to try to solve some of the problems highlighted in Chapter 2 thus customization was not a necessary component. For the same reason textual information visualized in the Evolution perspective is not very reach: this is an issue that comprimeses readability of the metrics shown and focused entities.

Chapter 4

Validation and Testing

4.1 Case Study A: Medium Size Repository

The first case study that will be treated has been is the SVNKit ptoject. It is a medium sized repository that is not completely representative of projects of its size but containt some interesting issues that will be visualized by the framework. The repository (at the time of replication) had more than 5000 revisions ranging on a time span of bout 1500 days and containing more than 300 Java Sources.

4.1.1 Performance

Before submitting the framework to a series of validation tests (some of them comparative) we show here some performances measures. We are interested in measuring time consumed to perform the overall repository analysis as well as the analysis performed on seuqential working copies that make up snapshots of the software evolution.

4.1.2 Validity

Consistency of data gathered has been verified through the X-Ray plugin [http://atelier.inf.unisi.ch/ malnatij/] that has been executed on working copies taken at the revisions listed below. Then we show some screenshots capturing particular phenomena in the software evolution that where expected to be catch by the framework.

Activity analysis Additions: 621, Deletions: 43, Modifications: 248 Removals: 0. Total activity is then: 912.

Results Visualizations show symptomatical (and expected) correlation between high values of the LoC per method ratio and activity performed on some entities, as well as correlation between an high number of methods in very active entities both in commits and LoC activity. However reliability of visualizations (and therefore of consequent correlations) strongly depends on coherence between temporal snapshots of the software history and commit frequency of the developers. When this is not the case we observe artificial entities behaviors hiding the real activity.

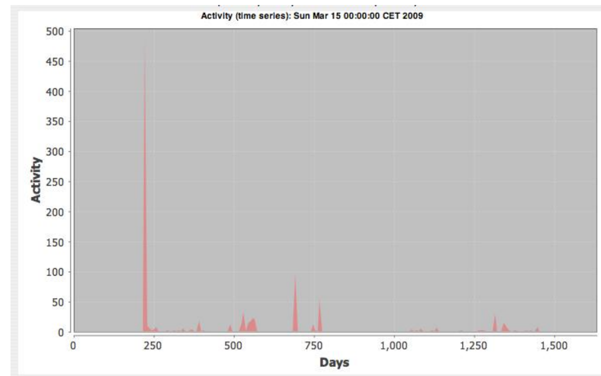

Figure 4.1: A view of the entire SVNKit project's activity.

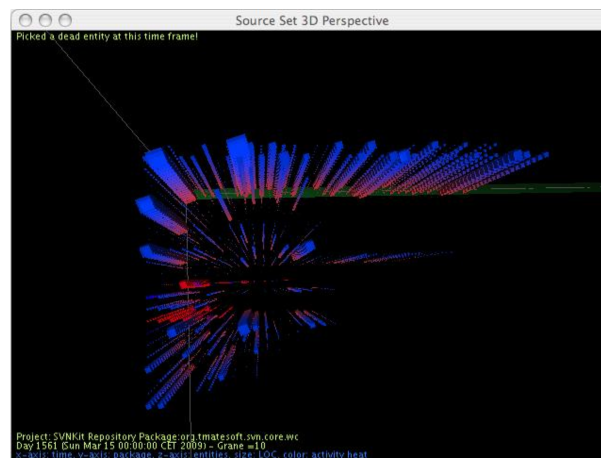

Figure 4.2: A view of the entire SVNKit project. Time dimension goes toward the observer.

revisions	I pass [min]	II pass [min]	Caching thread [min]
500	0.01	0.2	0.02
1000	0.01	0.2	0.02
2000	0.06	1.1	0.08
3000	0.1	2.1	0.12
4000	0.2	4.5	0.15
5000	0.5	6.3	0.16

Table 4.1: Framework performance analysing SVNKit's repository with a time granularity of 7 days (i.e. a time unit is 7 days and snapshots are taken at each time unit)

revisions	I pass [min]	II pass [min]	Caching thread [min]
500	0.01	0.1	0.02
1000	0.01	0.1	0.02
2000	0.01	0.2	0.05
3000	0.04	0.5	0.07
4000	0.06	1.3	0.07
5000	0.1	2.0	0.14

Table 4.2: Framework performance analysing SVNKit's repository with a time granularity of 21 days (i.e. a time unit is 21 days and snapshots are taken at each time unit)

revisions	total files	source files	Java total WLOC	Days of Life	author's number
500	12	193	130	131	3
1000	472	305	32757	250	10
2000	472	308	39560	332	11
3000	472	308	38411	1235	11
4000	567	308	39838	1421	11
5588	584	308	40371	1550	11

Table 4.3: Repository information gathered in the first pass

revisions	total files	source files	Java total WLOC	Days of Life	author's number
500	12	193	112	?	?
1000	472	305	30955	?	?
2000	472	308	37172	?	?
3000	472	308	36002	?	?
4000	567	308	36903	?	?
5588	584	308	39817	?	?

Table 4.4: Snapshots analysis using X-Ray [http://atelier.inf.unisi.ch/ malnatij/] as a verification and validation tool

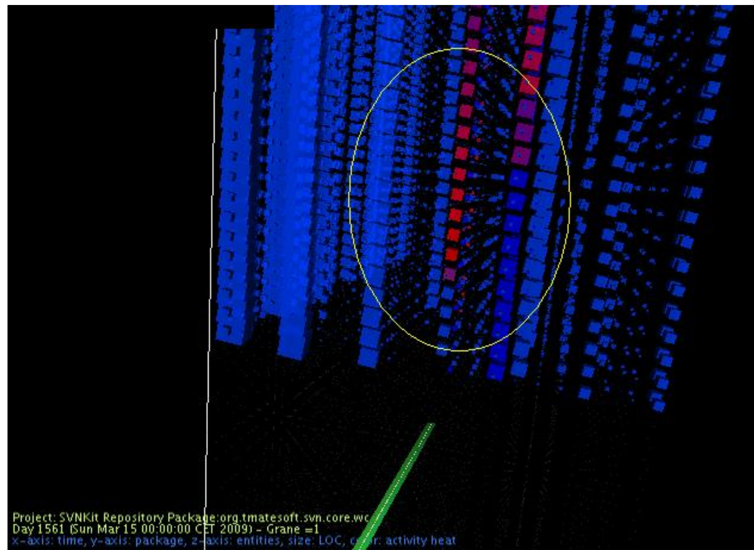

Figure 4.3: A look at a very active entity in package in EVOLUTON 2 view `org.tmatesoft.svn.core.internal.util` revisioned almost at every snapshot taken on the repository. Notice sourrounding entities belonging to other packages that where committed finished at the beginning of the project and that has no more been modified.

Chapter 5

Conclusion

The validation part has proved us that METROX Frameworkcan be used to provide some intuitive visualuzations of a software's evolution. However we are more convinced that the framework achieve is goal when used in parallel with other visualization tools.

Although not every expectation has been fulfilled, correlations highlighted in Chapter 4 show reltionships between an entity's complexity at the design level (as well as WLOC) and the activity performed on that entity. Entities containing a medium or a high number of methods and LOC ar more prone to revisioning than others. The particular structure of the reposiotry main branch however highlights a major problem that we discussed in Chapter 2: since many sources where committed already completed at the beginning of the repository's history, much information about their revisioning has been lost. Thus visualization of entities contained in these sources is not realistic because it is too constant in time, proving that analysis of the evolution between snapshots (above all the very early ones) is necessary in some, if not all, cases.

Further extensions of the framework will try to take in account the problem of changes between commits. They will also extend the visualization set to a more specific analysis of the design, gathering more object-oriented metrics.

Bibliography

[1] M. Jazayeri, H. Gall, and C. Riva. *Visualizing software release histories: The use of color and third dimension.*, In ICSM99 Proceedings (International Conference on Software Maintenance), IEEE Computer Society, 1999.

[2] M. Lanza *The evolution matrix: Recovering software evolution using software visualization techniques*, in Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution), 2001.

[3] M. Lanza and S. Ducasse.*Understanding software evolution using a combination of software visualization and software metrics.*, In Proceedings of LMO 2002 (Langages et Modles Objets), pages 135−149, 2002.

[4] M. Lanza and S. Ducasse. *Polymetric views – a lightweight visual approach to reverse engineering.*, IEEE Transactions on Software Engineering, 29(9): 782795, Sept. 2003.

[5] M. Lanza. and R. Robbes. *A Change-based Approach to Software Evolution*, in Electronic Notes in Theoretical Computer Science 166, pages 93109, 2007.

[6] M. Lanza and C. Mesnage *White Coats: Web-Visualization of Evolving Software in 3D*, 2005.