POLITECNICO DI MILANO

Visualizing Software Release Histories: The Use of Color and Third Dimension

carried out at
Technical University of Vienna
Information Systems Institute
Distributed Systems Group

under the guidance of

Prof. Dr. Carlo Ghezzi
Prof. Dr. Mehdi Jazayeri
Dr. Harald Gall

by

Claudio Riva

Milan, June 1998 _____

Ai miei genitori

Non è perché le cose sono difficili che non osiamo. E' perché non osiamo che le cose sono difficili.

Seneca

Se di affetti veraci in voi non è
sentito impeto primo;
se dall'anima vostra non prorompe
a dominar dell'uditorio i cuori
col vigore dei succhi saporosi
tratti alle scaturigini profonde,
vano è sperare d'acciuffarlo altrove
quel suasivo fascino del Verbo.
Sedete pure lì per giorni e giorni;
intrugliate in intingolo impastato
con avvanzi sottratti alle altrui mense;
soffiate pure sodo
a spremer miserevoli fiammelle
su dal mucchietto della vostra cenere.
Riuscirete a stupefare soltanto
bamboli e scimmie, se gli osanna loro
al palato vi tornano graditi.
Ma non potrete mai stringere al cuore
un altro cuore, se dal vostro cuore
non provenga l'impulso alla parola.

Goethe

... per me l'unica gente possibile sono i pazzi, quelli che sono pazzi di vita, pazzi per parlare, pazzi per essere salvati, vogliosi di ogni cosa allo stesso tempo, quelli che mai sbadigliano o dicono un luogo comune, ma bruciano, bruciano, bruciano come favolosi fuochi artificiali color giallo che esplodono come ragni attraverso le stelle e nel mezzo si vede la luce azzurra dello scoppio centrale e tutti fanno "Ooohhh!".

Kerouac

I would like to thank all the people who supported me in creating this Master's Thesis. Special thanks to Prof. Carlo Ghezzi and Prof. Mehdi Jazayeri who gave me the great opportunity of working at the University of Vienna and assisted me with their experience and interest. Special thanks to Dr. Harald Gall for his advice and guidance during the development of the thesis. Special thanks also to all the colleagues of the Distributed Systems Group for their help and friendships.

Profound and sincere gratitude is for Elke Neuwirth who has been close to me since this thesis had been a small idea and always encouraged me to continue. She read the whole thesis as last minute reviewer.

I would like to thank the family Neuwirth for its hospitality and courtesy in introducing me in Austrian culture and food.

Thanks to all my Italian friends with whom I spent nice moments by e-mail and make me feel at home.

Thanks to my family, particularly my parents Francesco Riva and Adele Formenti, to which this thesis is dedicated, who trusted and motivated me during my student career, that this thesis is the final step.

Vorrei in primo luogo ringraziare il Prof. Carlo Ghezzi e il Prof. Mehdi Jazayeri che mi hanno concesso la grande opportunità di lavorare all'università di Vienna e mi hanno assistito con la loro esperienza e il loro interesse. Ringrazio in modo speciale il Dott. Harald Gall per i suoi consigli e la sua guida durante lo sviluppo della tesi e tutti i colleghi dell'istituto per il loro aiuto e la loro amicizia.

Esprimo la mia più profonda e sincera gratitudine per Elke Neuwirth che mi è stata vicino fin da quando questa tesi era solo una piccola idea e mi ha sempre incoraggiato a continuare. La ringrazio anche per aver letto l'intera tesi.

Ringrazio la famiglia Neuwirth per la loro ospitalità e cortesia, e per avermi introdotto nella cultura e nelle tradizioni culinarie austriache.

Grazie a tutti i miei amici italiani con cui ho passato piacevoli momenti via e-mail e che non mi hanno mai fatto sentire lontano da casa.

Un caloroso grazie alla mia famiglia, e in particolar modo ai miei genitori Francesco Riva e Adele Formenti a cui questa tesi è dedicata, che hanno avuto fiducia in me e mi hanno motivato durante la lunga carriera scolastica di cui questa tesi rappresenta l'ultimo passo.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Software systems must evolve to satisfy new demands of the customer or to adapt to a changed environment. Often the evolution makes the systems large, complex and difficult to evolve. A critical point of complexity is reached when any additional extension would require exorbitant costs and a lot of changes through the whole system. At that point the system requires to be examined or restructured. The examination of the historical evolution can uncover potential shortcomings in the structure of the system and identify the blocks that need restructuring or reengineering. The purpose of the present work is to develop a three dimensional visual representation for examining the release history of the software. The structure of the system is displayed by 2-D or 3-D graphs. The historical information is displayed by taking time as the third dimension. Colors are used for displaying historical changes in the system. Visualization helps the human understanding and enables the humans' pattern matching skills. By means of multiple views and navigation support the user can navigate in the 3-D space to browse system information, to find interesting patterns and to discover unknown relationships and dependencies among system components.

Chapter 1

INTRODUCTION

The present Master's Thesis is in the context of the European project ARES (Architectural Reasoning for Embedded Systems, ESPRIT Project #20477) whose main objective is to enable software developers to explicitly describe, assess, and manage architectures of embedded software families. To reach this goal the project selects, extends or develops a framework of methods, processes and prototype tools for incorporating architectural reasoning along the life-cycle of embedded software family. Results of this project will help to design reliable systems with embedded software, that satisfy important quality requirements, evolve gracefully and may be built in-time and on-budget.

The work has been carried out at the Distributed Systems Group of the Technical University of Vienna. The group is directed by Prof. Mehdi Jazayeri.

The research in the department is focused toward the development of technologies that enable the construction of advanced distributed applications. The group is specially interested in information-intensive groupware. Such software systems support distributed workgroups involved in knowledge- and information-intensive activities. An example is a team of educators jointly developing material for a course. Each person brings special expertise to the group but face-to-face meetings are inadequate because access to a large amount of information is necessary to resolve most issues that come up in discussions. The computer support must provide group access to local information, such as the contents of other related courses at the same University, past examinations, and student inquiries, as well as non-local information such as on-line libraries, newsgroups, and both national and international forums devoted to the subject. The system must also support the group development of subject-specific laboratory experiments, simulations, and so on, which may require computation, visualization, and animation. The interaction among team members is often asynchronous but real-time interaction is sometimes necessary when reviewing a particular workproduct. Final course products may be in the form of text, electronic, audio, video, or a combination. Such information-intensive activities characterize many other areas such as a team of engineers involved in software development. The engineers need access to source code, design and requirements documents,

and regulatory requirements that may be available elsewhere. Another example of such information-intensive activity is exhibited by a group of analysts engaged in competitive analysis for a new product. Research investigates the requirements and component technologies for information-intensive groupware: software components, network services, programming languages, and software engineering technologies.

In the context of the project ARES Gall et al. [Gall97] developed a methodology for assessing the structure of a software system. The innovative and original approach of the methodology consists of the use of the historical evolution of system structure. Such structural assessment can uncover potential shortcomings and identify modules that should be subject to restructuring or reengineering.

The methodology has been developed based on previous works and its goal was to investigate and verify the salient features addressed in the methodology.

1.1 Goal of this Thesis

Software systems must evolve to satisfy new user's demands and to adapt to the changing scenario. New features required by users and new by developed technologies have to be implemented in the software system. Modifications, corrections and enhancements are applied to the system during the years. A critical point is reached when the complexity and the size of the system make further development difficult: new releases take longer and changes require exorbitant costs. This phenomenon is called software aging: software systems are functionally evolving but structurally deteriorating [Parnas85]. To avoid such a situation reverse engineering technologies help in maintaining existing software systems. The objective is to examine and manage existing software to increase its overall comprehensibility for maintenance, reuse or re-development.

The work of Gurschler [Gursch96] proposed a methodology for examining the structure of an existing system. The software system is a Telecommunications Switching System. The investigation involves twenty system releases that were delivered over a period of about two years. The system structure of several releases are stored in a database. The work of Gurschler is based on the data stored in the database. He examined the historical evolution of the system structure to discover potential shortcomings or to identify modules that require restructuring. Due to the huge size of the system, the examination has been realized extracting statistical information from the database. The examination produced useful observations about the whole system and discovered an anomalous subsystem which reveals a different behavior from the whole system.

The volume of data contained in the database causes a problem: how can useful information that is hidden within the data be extracted ? The ability to

extract and to understand these information is the key to increase the comprehensibility of the system.

The purpose of this Master's Thesis is the development of a visual representation for supporting the examination of the historical evolution of the system structure. Visualization is an emerging technology for extracting information from large and complex data sets. Presenting the data in a pictorial form the process of extracting information moves from being a cognitive task to being a perceptive task. The main objective of the present work is to investigate the possibility of visualizing the information contained in the database of the software releases for improving its examination process. It is the intent of this thesis to show that a visual representation of data can help to extract useful information. It is not an intent of this thesis to interpret the extracted information.

The development of the visual representation investigates the use of two graphical technologies: color and third dimension. Color can encode additional data in traditional graphs. The main advantage of color is that it stimulates the human visual system, enables the humans' pattern matching skills and also engages the user because of the increased visual pleasure. For software data visualization the third dimension is a new and rapidly developing area. The main advantage is that in a natural way more data can be packed onto the screen without overloading it. The information content (the ratio of information to pixels) can be increased by a factor of 10. It's difficult, however, to find 3-D representations of shapeless data like software information. In this thesis the third dimension is used for two intents. The former is to give a 3-D fashion to traditional hierarchical representations and the latter is to visualize the historical information

1.2 Organization of this Thesis

This Master's Thesis is divided into ten chapters.

The first chapter is the introduction. It describes the purpose and the organization of this thesis.

The second chapter discusses background concepts that are useful for the rest of the thesis. It describes the dynamics and consequences of the evolution of software systems. It also presents and overview of the reverse engineering technologies.

The third chapter covers the introduction to the case study. The case study is a Telecommunications Switching System. The chapter describes the system structure and the data that are available for this thesis.

The fourth chapter describes the methodology for analyzing the Software Release History as it first formulated in [Gursch96] [Gall96].

The fifth chapter is a review of the state of art that is relevant for this thesis. It presents two tools which have been developed for examining large

software systems. The first part is an overview of the tools. The second one is organized by ideas and describes their advantages and defects.

The sixth chapter explains the problem addressed in this Master's Thesis. The chapter is divided in five sections. At first the context of the problem is introduced. The second part presents a concise description of the main question that this thesis tackles. Motivations for solving the problem and purposes are presented in the next two sections. Finally, the last part discusses the advantages of the solution proposed.

The seventh chapter presents the approach adopted in this thesis for solving the problem stated in the previous chapter. The chapter is divided in four sections. The first section explains the requirements of the visual representation. Then the adopted approach is carefully described. The third sections shows how the visual representation is used for examining the Software Release History. In last section the advantages of the approach are summarized and discussed.

The eighth chapter describes the tool that has been developed for supporting the graphical visualizations explained in the previous chapter. The purpose is to show the original ideas that have been applied for its development, and not a description of its functionality.

The ninth chapter discusses some conclusions. Three topics are covered: conclusions about the present work, summary of contributions of new knowledge that this thesis makes and ideas for future research that have been produced during the development of this thesis.

The tenth chapter lists the publications that have been referenced in this Master's Thesis. The references are ordered alphabetically by author surname.

Appendix A is a short resume in Italian of the whole thesis.

Appendix B describes the mathematical concepts used in this thesis for visualizing graphically hierarchies.

Chapter 2

BACKGROUND

2.1 Large Software Systems

The present work is focused on *Large Software Systems*. A *system* is "(…) a grouping of interrelated components that, together, form a unified whole that accomplishes a specific purpose or a set of purposes (…)" [Encyc94]. The definition underlines that a *purpose* is the grouping factor of the set of components and that the components, through their interrelationships, fulfill the system's purpose. A *software system* is a specialized system whose components are made in software. Such a generic definition includes all kinds of software programs, spanning from a set of routines which just implements an algorithm to a more elaborated system.

The term *large* is used to classify a particular sub-class of software systems. A standard definition of large software system doesn't exist in literature, some properties characterize them. Belady and Lehman define a system large "(…) if it reflects within itself a variety of human interests and activities. " [Lehman78]. They use the concept of *variety* as the root cause of the largeness of a software system. Attributes such as number of instructions or modules, resource demands, amount of documentation are indicators of the variety. A large software system essentially cannot be understood by a single individual, it needs an organized group of people to design, implement, maintain and enhance it. The next section describe the evolution process of software systems and shall clarify when the nature of largeness appears in a system.

2.2 The evolution process of a software system

The life cycle of a software system has been largely investigated in the past years. Investigation shows that the development of a software product is not a static process and it doesn't stop after delivering the software system. New demands make the system change to adapt to the changing environment.

Terms like *maintenance*, *evolution*, *software aging* are used in literature to characterize this behavior. This section explores their meaning and their consequences.

2.2.1 The maintenance phase

Once the product has been delivered to the client, the software product enters the maintenance phase. Any changes to the product are done to maintain the system. The word *maintenance* has been borrowed by mechanical engineering and in software engineering context it has a different meaning. Software doesn't deteriorate, of itself, and so it doesn't need to be maintained in the traditional sense. The purpose of this phase is to keep the software operational and responsive [Martin83]. The standard IEEE definition for software maintenance is:

> "The process of modifying a software system or components after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment." [IEEE93].

The maintenance phase is characterized by the fact that all the modifications to the system are carried out after delivery of the product. The input is an existing product that somebody is using. This is the peculiar characteristic of maintenance. A person or a group of people are using the system for the purpose that it was developed. In literature those persons are called *users*. Users can formulate comments, suggest improvements, report defects. In other words they can be satisfied or not, and their satisfaction can change with time. The satisfactory level is a feedback for the system's developers. Lehman and Belady classify software system according to the use that developers make of the feedback [Lehman82]. In the present work the focus is on systems in whom the feedback is considered by developers and is a reason for modifying the system. Such systems are named *embedded* because they are embedded in their operational environment. Therefore, the user's feedback is an input to the maintenance phase. Several reasons for changing a software product after being delivered are summarized here:

- New demands and requirements of clients must be satisfied.
- The performance and the quality must improve to have a competitive product.
- Defects and faults have to be removed.
- Data formats, hardware configuration, interfaces to other programs have to be updated.

Maintenance is defined as the process of modifying the system to satisfy a specific purpose. Swan and Schach classify the maintenance process in four major kinds of works [Swan76] [Schach96]:

- Corrective maintenance (20%), which is the process of correcting any residual errors discovered after software is in use. For instance, specifications errors, design errors, coding errors, documentation errors, or any other types of errors. It is worthwhile to notice that only 20% of the effort is spent on corrective maintenance.

- Adaptive maintenance (25%), which is applied when changes are made in order to react to modifications in the environment in which the product operates. For example, a product have to change if it is ported to a new compiler, a new operating systems or a new hardware. Or changes in the data format (the year 2000 problem, for instance), standards, protocols. Adaptive maintenance is not a direct request of the client but it is externally imposed on the client.

- Perfective maintenance (50%), which includes all the enhancements that are requested by the client or by the user community. These changes are made to improve the effectiveness of the product such as additional functionality, performance, user interface, documentation style.

- Preventive maintenance and others (5%), which improves future maintainability and reliability and provides the basis for future enhancement.

The percentages are based on the Lientz and Swanson study of 487 software development organizations and represent the distribution of different kinds of software maintenance activities which these authors saw in the surveyed organizations. The numbers are consistent with other studies [GAO81] [Fjeld79].

Maintenance has the peculiar characteristic that it incorporates aspects of all the other phases of the software development process. It influences all the components and it's made on existing code The major maintenance tasks are adaptive and perfective maintenance. To perform these tasks, the maintenance programmers must go through all the phases of requirements specification, analysis, design, implementation and integration taking the software product as starting point. Some changes need additional modules, and they have to be implemented and integrated in the system. All these activities have to fit into existing data and structural constrains, and much of the design effort is devoted to explore the current programs and to find out how to integrate the new features and what their impact is on existing functions.

Conventional software engineering tends to neglect this part of the product lifetime and to assign it a small role after the delivery of the system. The focus is on the design and the implementation of the product [Schach96].

Studies on large system show that almost half the time of the life cycle of a software product is spent during the maintenance phase [Lientz80]. An HP research estimated that 40% to 60% of the cost of production was directly related to maintenance and that 60% to 80% of research and development

personnel were involved in maintaining 40 to 50 million of lines of code [Pearse95]. So the bulk of the lifetime cost is related to this last phase, where one must live with the product previously delivered.

2.2.2 Maintenance as evolution

The works of Belady and Lehman [Lehman76] [Lehman85] suggested that the maintenance phase should not be distinguished by the development phase but they should be considered as closely related activities and parts of the overall software life cycle. They use the term *evolution* to describe the changes that the program undergoes from beginning to end of its life.

In biology science the theory of evolution describes the development of more complicated forms of life (plants, animals) from earlier and simpler forms. The development of a software system is analogous to the evolutionary process of any complex systems.

The development of a new product cannot yield a first release that is perfect for the intended application. The user's requirements are sometimes difficult to be understood or guessed. The environment changes during the development. Satisfactory levels of system attributes (such as correctness, reliability, performance, functionality) are all achieved iteratively through several release of the product. The concepts, algorithms and techniques that implement the program change as the design proceeds. All these motivations involve that the development process must undergo through successive phases of refining of a rough product.

During the in-service phase the interaction with the environment stimulates to modify and to enhance the system. Users' feedback and demands encourage the addition of new capabilities, advances in technology encourage to implement them and advances in hardware to support them. Economy factors require a competitive, modern and efficient product. All these developments need continuing revisions and modifications of the software system.

Software system evolves, release after release, adding new functionality, removing others, re-designing, re-implementing and replacing if the complexity is too high.

The works of Belady and Lehman [Lehman85] point out that the need of continuous increase in the potential power of the system is intrinsic in the nature of computer applications, evolution must be accepted as "a fact of life".

Recent model for the software life-cycle [Skram92] and for the software development process [Schach96] incorporate the evolutionary behavior of software products. The slogan "Design for change" and the paradigms such as *information hiding*, *abstraction*, *data hiding* or *object-oriented* are all means to build the maintainability in the products from the very beginning of the development process.

The evolution of software systems occurs through successive releases or sub-releases of the system itself [Lehman82]. The system *release* is a mechanism for the controlled implementation of changes and their transmission to users. Changes become integral part of the system when the release is delivered and no other changes are allowed after the delivery. Successive changes are integrated in the next release of the system. In this way the release information can locate uniquely a well-defined implementation (source code and documentation) of the system. A release may consists of a single change or of a number of corrections, enhancements and additions. Whatever the kinds of change are, a release freezes the state of the system. So the evolution of a software system is recorded by its release information.

2.2.3 The aging of software

Evolution is an intrinsic behavior of software products. The user's satisfaction of a release of the system is relevant for a limited period whose duration depends on the foresight of designers and the rate of change in the operational environment [Lehman82]. It relies on designer's capability of predicting the future, i.e. new scenarios, new changes, new demands. This capability is intrinsic limited for humans[Parnas94].

The need of maintaining an existing system arises from the necessity of extending its lifetime. The development or the purchase of a new system has higher costs than maintaining the old one or often it's not possible because there are no other similar products in commerce. An example is described in [Roch93]: the company was forced to maintain its existing system. The assumption behind this principle is that the longer software lives, the better it is.

Evolution is used to extend the lifetime of software systems. Modifications and enhancements are used to adapt the systems to the new demands. But the evolution cannot continue indefinitely. Studies [Lehman85] show that the costs of changing grow with respect to system's age. In literature this phenomenon is called *software aging*.

Parnas [Parnas94] identifies two causes of software aging. The first is the failure of the software to satisfy user's needs. Over the years user's expectation increases as technology evolves, the most advanced technologies become the user's standard requirements for the systems. New paradigms are preferred to old ones and expected in the products. If a software system is not frequently updated to follow new paradigms, users become dissatisfied and change to a new product as soon as the benefits outweigh the costs of conversion. When this happens the system has aged.

The second cause is the result of the changes that are made on the system. The process of evolving a system to extend its lifetime comports to modify and enhance it. These modifications alter the structure of the system in a way that can deteriorate the structure itself. Aging arises from deteriorating the structure. This concept and its consequences are explained below.

Modifying or enhancing a piece of software relies on the possibility of understanding the basic concept that the designer implemented with the software itself [Parnas94]. Understanding this concept allows one to know how to alter it and which are the effects of the alteration on the others components. Modifications of pieces of software that are not included in the designer's anticipations and are not well understood lead to degenerate the structural integrity of the system. This means that after several changes the designer and the person who made the changes are not able to understand the modified product. The structure can degenerate at a point that, if no other actions against aging are taken, it terminates the life of software.

The problem of modifying existing software arises from the presence in the system of *old code*. The oldness of code isn't necessarily related to the time when it was written. Time is one of several factors, other factors are [Corbi89] [Bennet91]:

- Documentation is nonexistent, incomplete or not updated. Often, deadlines or manager's pressure cause to neglect documentation. Changes are quickly documented or not at all and they are not integrated in the existing documentation.

- Poor design or old design for the new scenario to whom the system is adapted.

- The use of obsolete programming language or programming language that doesn't easily communicate the program structure, data abstraction, types, functions. For example unstructured programming language.

- Impossibility to find the persons who wrote that code, due to the high rate of personal turnover.

- The manner of maintaining the system. As described in Section 2.2.1 the maintenance phase is often neglected, undertaken, or considered an after-sales service by managers.

All these factors make difficult or at most impossible the task of understanding, redesigning, modifying, debugging or rewriting existing code, in other words the task of evolving the system.

Software whose structure has degenerated becomes expensive to update [Keller96]. Changes take longer and introduce new bugs. Even small changes can require large efforts because their impact on the system has to be analyzed in depth to avoid incompatibility. New releases are difficult to deliver, they take longer and the system become more and more non-maintainable. Parnas [Parnas94] points out three consequences of software aging:

- Increased difficulty to keep up with the market. As the size and the complexity of the system increase, changes become more complex because they require to change more code and more time has to be

spent to find the code to change. As a result, it's difficult to add new functionality and the system cannot evolve quickly enough to satisfy its customers who may eventually switch to younger products.

- System decreases its performance. As the size gets bigger the system requires more resources on the computer and the system responds slowly. An old design of the system or a poor design can also affect the performance because the software is subject to scenarios not visualized in the original design. As a result, users may switch to a product whose performance is better and that has been designed ad hoc to support the new features.

- System decrease its reliability. Changes for enhancing or removing bugs that are not well understood can introduce new bugs and problems in the system in the new releases with the user's dissatisfaction. A major concern of maintenance programmers is to avoid of introducing more problems than are solved by modification.

2.2.4 Coping with software aging

Aging is a problem that can affect the evolution of software products. Measures to prevent it and to limit its effects have to be used to avoid the decay of the system. Belady and Lehman [Lehman71] observed that *progressive* changes in the system require *anti-regressive* efforts to keep the complexity manageable. This means that changes in the system have to maintain the integrity and consistency of the system itself.

As described in Section 2.2.2, recent studies try to help programmers to cope with aging from the begin of the development process. Other technologies or guidelines such as *updated documentation*, *reviews*, *change and bug reports* should be preventive measures to avoid aging during maintenance [Schach96] [Bennet91].

Difficulties arise when dealing with existing aged systems. Preventive measures may not have been used, changes couldn't be documented, documentation is inconsistent or incomplete, the original design is obscured by changes. The only trustable information is the code but the use of obsolete programming language could make it unusable. New approaches have been studied or are developing to deal with aged system. The next section focuses on this scenario.

2.3 Reverse Engineering Technologies

Software reverse engineering is a recent [Ulrich90] technology whose purpose is to help in maintaining existing software systems. The basic idea is to examine and manage existing software to increase its overall comprehensibility for maintenance, reuse or new development. The term *reverse engineering* and related terms such as *software reengineering, restructuring*, *re-documentation*,

design recovery, are used to define different aspects of the technology. This section defines their meaning and their purpose.

2.3.1 Definitions

The definitions used in this work were first presented in the paper of Chikofsky and Cross [Chikof90] and subsequently maintained by the Taxonomy Project of the IEEE Computer Society TCSE's Subcommitee on Reverse Engineering [Encyc94].

The definitions make the following assumptions:

- ✓ The presence of a subject system such as a single program or a complex set of interacting programs.

- ✓ The presence of an orderly life-cycle model for the software development process. The typical life-cycle can be decomposed in three stages: requirements, design and implementation. Requirements is the process of specifying the problem being solved, including objectives, constraints and business rules. Design is the process of developing and specifying a solution needed to meet the requirements. Implementation is the process of coding, testing and delivery of the operational system. These activities represents different levels of abstraction of the system. Requirements are the highest level of abstraction, they specify what the system have to do. Design specify how the system can implement them without describing implementation details and constraints. And the implementation yields the most concrete representation of the system, the system itself.

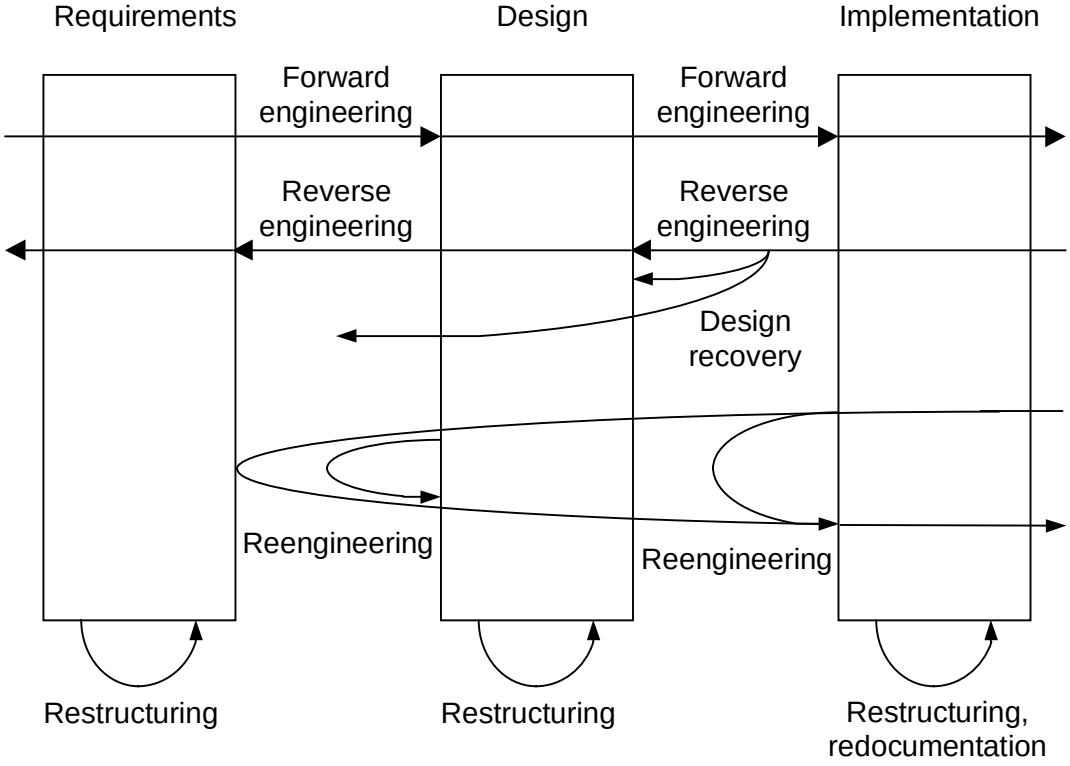

Figure 2-1 Relationships between terms. Adopted from [Chikof90].

The life cycle model and the relationships between terms being defined are illustrated in Figure 2-1.

Forward Engineering

Forward engineering is the traditional process of moving from high-level representations and logical, implementation-independent designs to the physical implementation of the system. It follows a sequence from the analysis of requirements through design, and finally to implementation. It is a movement from an high level of abstraction to a lower level.

Reverse Engineering

Reverse engineering is the process of analyzing a subject system in order to identify the system's components and their interrelationships, and to create representations of the system in another form or at an higher-level of abstraction. It is a movement from low level of abstraction to an higher level. Reverse engineering is a process of examination, not a process of change. Its purpose is to increase the understandability of the system. This process of analysis can start at any level of abstraction or any stage of life-cycle. Related sub-areas are *re-documentation* and *design recovery*. Re-documentation is the creation or revision of semantically-equivalent representations of a subject system within the same abstraction level. The intent is to recover documentation that existed or should have existed. Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher-level abstractions beyond those obtainable directly by examining the system itself.

Restructuring

Restructuring is the transformation from one representation to another at the same relative abstraction level, while preserving the subject system's external behavior, in particular its functionality and semantics. A restructuring transformation is often altering the code to improve its structure in the traditional sense of structured design. This process can be applied to any level of abstraction. Its purpose is to improve the physical state of the subject system with respect to some preferred standard or to adjust the system to meet new environmental constraints. It is often used as a form of preventive maintenance.

Reengineering

Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. It includes some form of reverse engineering (to achieve a more abstract description) followed by form of forward engineering or restructuring. Implied in this term is the possibility of change in essential requirements, rather than a mere change in form. The purpose may be to include modifications with respect to new requirements not met by the original system. When this process is not included reengineering may look like a restructuring process.

2.3.2 Motivation of Reverse Engineering technologies

Reverse Engineering technologies have been proposed as an answer to many of the problems of maintaining and evolving existing software systems. The need of maintaining and evolving such systems arises from the value they have acquired for many companies and institutions. These systems are labeled *legacy systems*. In literature there is no standard definition of legacy systems but a set of properties which qualify them. The common aspect of these properties is that they are a valuable product for a community of people. The properties are [Encyc94] [Roch93] [Gall96]:

- Huge investments of resources for a company or institution.
- Long in-service time, i.e. a range of 10 years or more.
- Critical component for the success of the respective organizations (e.g. banks, insurance companies, military institution, government offices, software companies, manufacturing companies, etc.)
- Worthwhile or not-trivial knowledge added during years.
- Impossibility of replacing or re-building because of costs or technical reasons.

These legacy system have evolved over the years and have become more complicated as described in Section 2.2. The need of extending their lifetime implies to increase their comprehensibility or to update the knowledge of them. The benefits of applying reverse engineering techniques to existing system are [Encyc94] [Arnold93]:

- Reduce an organization's evolution risk. The costs and the risks of building a new system are often less that maintaining the existing one [Roch93].
- Organizations can recoup their investments in software reusing existing system or parts of it.
- Organizations can become more flexible because software can be modified more quickly to accommodate business changes.

2.3.3 Purpose of Reverse Engineering technologies

According to the fonts used in Section 2.3.1 the main purpose of reverse engineering technologies is "(…) to increase the overall comprehension of a system for both maintenance and new development." [Chikof90]. Beyond this definition, six objectives have been defined:

- Coping with complexity. Reverse engineering tools have to deal with sheer volume and complexity of software systems.

- Generate alternate views. Different graphical representation of the system from different point of view can aid its comprehension. Reverse engineering tools facilitate their generation or regeneration.

- Recovering lost information. The continuous evolution causes to lose information about the system design. Reverse engineering is a way to recover information at higher level of the code itself, such as documentation or system designs.

- Detecting Side Effects and Analyzing Quality. Reverse engineering can help detecting anomalies or side-effects before user report them. It also provide additional observations beyond those obtainable with forward engineering.

- Facilitating Reuse. Reverse engineering helps detecting software components that can be re-used in other systems.

- Synthesizing Higher Abstractions. Reverse engineering allows the analyst to examine the system at an higher level of abstraction. High levels of abstraction can be synthesized from information collected at a detail level.

2.4 The Modular Structure of Large Software Systems

A large product that consists of a monolithic block of code results to be unmanageable even by the programmers who wrote it. An high level of abstraction is required to cope the complexity and size of large software systems. This level of abstraction is named *software architecture*.

Garlan and Shaw propose that a software architecture is "(…) a collection of computational components - or simply components - together with a description of the interactions between these components - the connectors. Graphically speaking, this leads to a view of abstract architectural description as a graph in which the nodes represent the components and the arcs represent the connectors." [Garlan93]. The definition focuses on two elements: components (such as functions, objects, modules filters, etc.) and connectors (function calls, events, pipes, etc.). The system is decomposed in these two elements. Shaw proposes several architectures that are often used in the system organization [Shaw89]: *pipes and filters*, *data abstraction*, *layered systems, rule-based systems* and *blackboard systems*. Such abstraction can simplify the design and the organization of the system itself.

Parnas [Parnas85] proposes a modular structure for large and complex systems. The structure is based on modules decomposed according to the information hidden. The main idea is to group the system details that are likely to change together and to separate the ones that can change independently. The only assumptions should appear in the interfaces between the modules.

System decomposition is not enough to cope complexity and size because the system software could be decomposed in hundreds or thousands of modules. Module decomposition has to be supported by another principle. Parnas proposes a modular structure where each module is a part of a tree hierarchical structure.

In the present work of thesis a software system is considered as being decomposable in modules. Any decomposition principle that leads to a set of modules is accepted without any distinction. It doesn't require that the system has a modular structure but it requires that a modular structure can be assigned to the software system.

2.5 Software Measurement

A software system is the product of a development process. To understand, control and improve the process managers and engineers have to measure the characteristics of quality. In literature *software measurement* is defined "(…) as a mapping from the empirical world to the formal, relational world." [Fenton96]. The definition of *software measure* is "(…) the number or symbol assigned to an entity by this mapping in order to characterize an attribute." [Fenton96]. A measure can be used to characterize some property of software code quantitatively. For example, the measure *Lines of Code* characterizes the property "size of software code" by associating a number (i.e., number of lines of code) with it. The measure *Number of Faults* characterizes the property "error proneness of the code" by associating a number(i.e., number of faults detected) with it.

Chapter 3

THE CASE STUDY

The present section describes the case study considered in this Master's Thesis. The same case study described here has been used in the related works [Gall97] and [Gursch96].

3.1 Overview

The case study examined is a product family of Telecommunication Switching Systems (TSS). This product family covers a wide range of utilization. The area of use ranges from small local switches in a fixed network to large international switches and switches for mobile phones. A TSS of the family includes the hardware as well as the software required to run the switching system. The evaluation of the present work only concerns the software of the system.

The TSS was first shipped in the early 1980s. At that time it was the first fully digital local switch. The switch was initially planned as a low-end local switch, then the switch exceeded its performance expectations and became a product family for a wide range of utilization.

The implementation of the software of the initial release was done in machine-specific low-level language. After a few years the language was gradually replaced. Many different languages such as Assembler, C and Basic have been used to code new parts of the system. Presently, the system is being developed using SDL [Sarma96]. The SDL programs are translated into C and then compiled with a standard C-compiler.

The system was under continuous development and several re-designs of the software and hardware have been done. The first delivery of the TSS in 1980 had 100.000 LOC (lines of code). In 1990 a typical TSS product consists of 3 MLOC (million lines of code). Today the size is about 13 MLOC. These figures already show some of the issues of such huge systems.

3.2 The structure of case study

The TSS consists of both hardware and software parts. The hardware part is common to all products of the family and it is not considered in this context. The software part is organized as a *layered system*. Layered systems are organized hierarchically with each layer providing service to the layer above it. Internal layers are usually hidden from outer layers, except for certain functions carefully selected for export [Shaw89]. The structure of the software is a tree hierarchy with four levels. The top level is the *system level*. It's based on the *subsystem level* (second level), *module level* (third level) and *program level* (fourth level). Figure 3-1 shows the level hierarchy graphically.

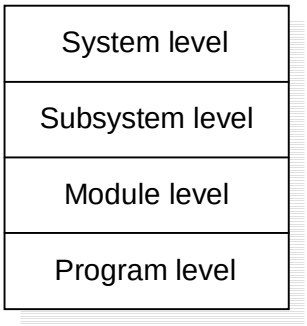

Figure 3-1 The four layer hierarchy of the case study.

Each level consists of one or more elements. Each element of a certain level is connected to one element of the higher level. The system level contains only one element representing the root of the tree. The element of the system level represents the TSS product.

The elements in each level are named corresponding to the names of the levels: the elements in the subsystem are named *subsystems*, the elements in the module level *modules* and the elements in the program level *programs*[1].

Figure 3-2 shows the software structure of the TSS. *Programs* are the smallest logical unit of this structure. They represents the algorithms. The algorithms of a *program* are implemented in source files. So one or more source files are associated to a *program* element. The tree hierarchy limits the visibility of the algorithms contained in the program level. For instance, a *program* can only be used by another *program* of the same *module*.

[1] To avoid confusion the names of the structure elements of the case study are written in italic: *subsystem*, *modules* and *program*. When they are written in normal characters they refer to the usual meaning of those words.

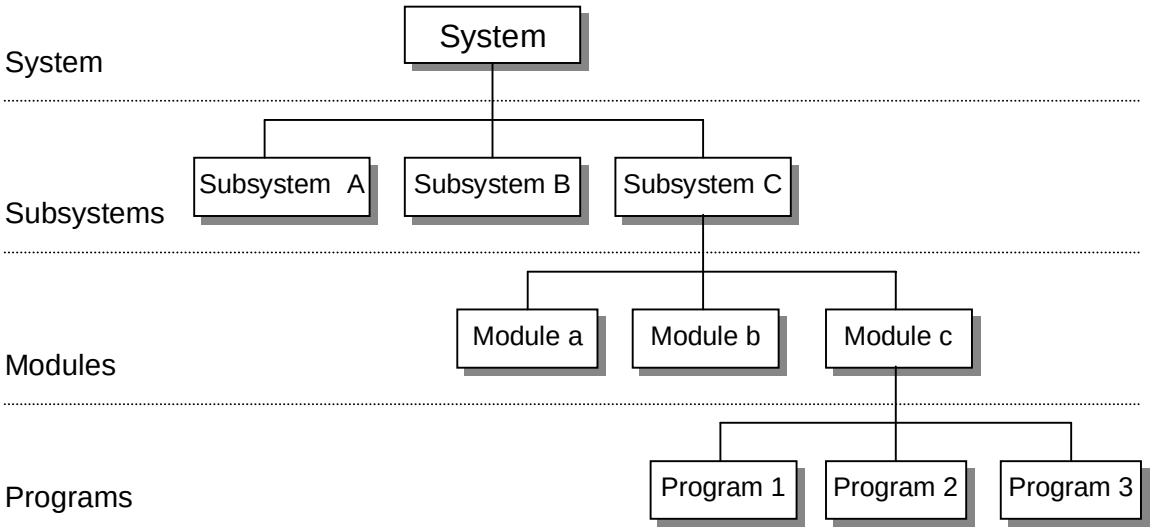

Figure 3-2 The software structure of the TSS.

3.3 The Software Release History

Telecommunications Switches are products that require extensive customization for different markets and applications. The customization of the system is mainly performed by making changes to the code in several parts of the system. Each customer receives a specially adapted system according to his/her requests. This flexible approach makes the customization fast and efficient, but the system becomes expensive to develop, test and maintain.

The system evolves through releases as described in Section 2.2.2. In two cases a new release of the system is delivered.

1. A new customization is requested the system is changed to satisfy it.

2. Improvement in the functionality or bug fixes require a new release to upgrade the system.

In both situations some parts of the code are changed. For instance, when a bug is found, only the *programs* affected by that bug are changed, the others are not changed. The new bug-fixed release contains the old *programs* of the previous release and the new corrected *programs*. In this way a generic release of the system can contain both new code and old code.

Due to the mix of code written in different releases a way for tracking the changes has been developed. The system element and *program* elements are

characterized by a version number. This method of tracking changes in the structure has been named *Software Release History*.

The releases are progressive numbered. Successive releases are numbered with increasing values. In literature this version number is named *system version*. In this way the system version has the role of time coordinate and the system is precisely individuated by its value. Each *system* element has the version number of the specific release. For example, at the release 5.3 the corresponding system element has system version 5.3. The Table 3-1 shows the version numbers of the case study. Twenty different releases are considered which represents releases over 21 months. Eight of these releases are major releases (release 1 through 8) and twelve are minor releases for releases 6 (6.01 through 6.12). Major releases represent substantial changes in the functionality of the system. Minor releases contain mainly bug fixes. The time intervals between major releases (1-3 months) are normally larger than between minor releases (15-30 days).

	1	**2**	**3**	**4**	**5**	**6**	6.01	6.02	6.03	6.04	>>
>>	6.05	6.06	6.07	6.08	6.09	6.10	6.11	6.12	**7**	**8**	

Table 3-1 Version numbers of the considered releases.

Each *program* element has its own version number whose numeration is independent of the version numbers of the *system* element, i.e. the system version. The number indicates uniquely the implementation of the *program* element. In each release of the system a *program* element appears with its own version number. Taking two different releases a *program* element can appear with the same version number or with a different version number; in the former case the same implementation of the *program* is used for the two releases, in the latter case the two different implementations are used. A *program* element can change several times its version number between two release of the system. The Table 3-2 shows an example. The first line contains the system versions. The *program* element A.c.1 is present in the releases 4 and 5 with the version number 2.0. This means that it is present with the same implementation in both the releases. Then it changes to the version 2.1 and the releases 6 and 6.01 contain the new implementation of it. The *program* element A.b.1 changes between the release 4 and 5 from the version 4.3 to the version 4.4. Between the release 5 and 6 it changes from the version 4.4 then 4.5 and then 4.6.

System version	4	5	6	6.01
A.c.1	2.0	2.0	2.1	2.1
A.b.1	4.3	4.4	4.6	4.6

Table 3-2 Example of version numbers.

This way of tacking the changes in the system allows to know the configuration of a release in terms of its *program* elements. A system version

uniquely identifies the elements of system at that release and the implementation of the system.

Note that the *subsystem* and *module* elements don't have any numeration because they represents an abstraction of the program level.

3.4 The Software Release History Database

The data regarding the software release history are extracted directly from the source code. During compile time preprocessors extract and store the information in a database. For each release stored, the database contains entries for elements at the system, subsystem, module and program level. Two valuable information are present: relations between various elements of the system (e.g. *module* c consists of *programs* 1, 2, 3) and the version numbers of the blocks (e.g. *program* A.c.5 has version number 2.3).

The database considered in the presents works is populated with 20 releases of the software product. Each release contains 8 *subsystems*, 47 to 49 *modules* and about 1500 to 2300 *programs*.

Chapter 4

THE SOFTWARE RELEASE HISTORY ANALYSIS

The software release history analysis has been proposed in the studies [Gall97] and [Gursch96]. This section describes the purpose and the approach of this method. The description is mainly focused on the cases study described in Chapter 3.

4.1 Motivation and Purpose

The aging phenomenon underlines that a software system is usually functionally evolving but structurally deteriorating. This comports that the continuous changes modify the original structure in a way that further development becomes difficult and requires exorbitant costs. As described in Section 2.3, the need of evolving a software system requires the use of reverse engineering techniques. A common task of these techniques is the examination of an existing system. The aim of examination is to increase the comprehensibility about the system itself in order to be able to operate on it. Dealing with large systems this task is often difficult to achieve because of their size and complexity.

The Software Release History Analysis proposes a method for evaluating the structure of existing large software system. As described in Section 2.2.2 a software system evolves through successive releases and the release history records its evolution. Each change or group of changes are recorded in a system release. The purpose of the method is to use the historical information to evaluate the evolution of the structure of the system. Such evaluation is useful to find out potential shortcomings of the system structure or to identify modules that need restructuring.

4.2 Overview of the approach

The Software Release History Analysis addresses the problem of examining large software system. The complexity and size of such systems discourage a code-oriented approach but require a more abstract way of examination. This approach is based on the examination of the *Software Release History*. The software release history tracks the evolution of the system in terms of changes from one release to the next one. *Observations* on this historical information can help in several ways:

- Discovering significant changes that lead the system to the existing structure.
- Discovering the changes that caused problems in the structure of the system.
- Discovering the significant events in the evolution of the system.
- Discovering anomalous behavior of some modules.
- Documenting the evolution.

The next two sections describe the software release history and the observations which are the base of the method.

4.2.1 Software Release History

A software system evolves through successive releases. Release after release the system evolves adding new functionality, removing others or changing the existing ones. This is the incremental process of developing a software system described in Section 2.2.2. The principle of this process is using the existing product at a specific release (for example, the latest one) to generate a new product slightly different which constitutes a new release of the system. In this way in each release the new code and the old one are mixed. The software release history is a method to track the mixture of old code and new code in different releases of the system.

The method assumes that a modular structure can be assigned to the system. This assumption arises from the nature of the case study used in the studies [Gall97] and [Gursch96] where the software release history analysis has been first formulated. The assumption is discussed in the Chapter 7. The modular structure allows to decompose the system in *modules* as described in Section 2.4. A hierarchical structure of modules is obtained by the decomposition (as described in Section 3.3). A module is a named component which is related to a piece or pieces of software in the system. For instance, a module could be a file, a group of files, a function, an algorithm, a class.

A *version number* is assigned to each module of the system structure. The IEEE definition of version is [IEEE93]:

"An initial release or re-release of a computer software configuration item associated with a complete compilation or recompilation of the computer software configuration item"

In our case the configuration item is a module. The implementation of the module can change in each release. Version is the term used to specify a particular release of the module. Version number is the unique number assigned to a specific release of the module. In this way a version number can uniquely identify the specific implementation of a module.

The releases of the system are numbered as well as modules. To each release a number is assigned. This number is named *Release Sequence Number* (RSN). This numeration is independent of the version number of the modules.

This approach allows to document and manage the implementation of the releases. Each release of the system is described by a structure of modules and by the configuration of version numbers of the modules.

Figure 4-1 shows an example of software release history. Two releases of the system are shown: release 1 and the successive release 2. For each release the structure of the system is shown. Each module is represented by a box which contains the name (left side) of the module and its version number (right side). Some changes happened between the two releases. Module B has been changed and the new implementation appears in release 2 with version number 1.0. Module C has been upgraded as well. Module A don't change and appears with the same version number. The sub-module a of module B has been removed in release 2 and module 3 has been added to the module c.

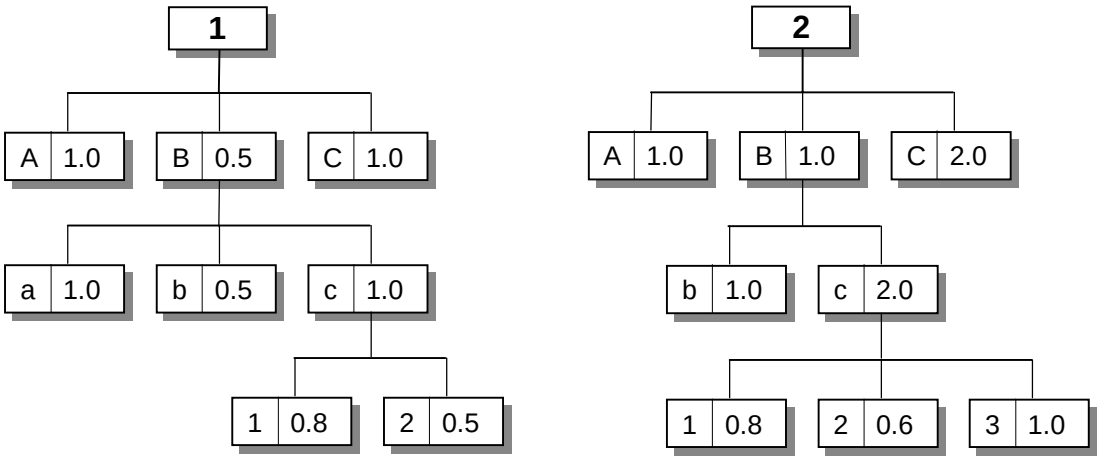

Figure 4-1 Example of software release history.

The information about the structure of the system and about the version numbers are stored in a database. Such database is named Software Release Database or Product Release Database. Other information about the module can eventually be stored in the database, such as size, maintainer's name, data of last change.

4.2.2 Software Evolution Observations

The software release history represents the evolution of the system at an higher abstract level that code level. Two steps have already been accomplished: decomposition of the system in modules to cope the complexity and the size of the system and assignment of version numbers and RSN to document the evolution. At this point of the analysis *observations* are used to extract useful information from the data.

The previous work of [Gursch96] addressed the problem of establishing a method for extracting observations from the software release history. The work also addressed the problem of testing the fitness of this method to identify structural problems. Two main problems have been solved:

- Which useful information can be extracted ? This problem has been solved setting up a list of queries. The answers to these queries should bring the information required to identify structural problems. The purpose is to find shortcomings in the structure and outliners which have significant different behavior compared to the rest of the system.

- Which conclusions about problems in the structure can be derived ? Once structural problems have been detected, hypotheses can be established. The hypotheses try to describe the reasons why the structure is evolving in the observed way. The verification of the hypotheses can lead to conclusions about problems in the structure of the system.

The queries are the method used to extract information from the system. Each query focuses on a structural property which could be an indicator of a problem in the system. The answer to a query should prove the existence of a problem or should locate it. Some queries are reported below. The purpose it to show how they are formulated and which is their aim. They are extracted from [Gursch96]:

"Are the changes distributed widely over the system ?"

"Which *subsystems* are changing more frequently than the average ?"

"Which *subsystems* are much larger that the average ?"

"Which *subsystems* where added ?"

Trying to answer these queries reveals where the problems are. In this way shortcomings or abnormal modules can be identified.

The described method for extracting observations has been applied to the case study reported in the present work. Several useful considerations have been produced about the whole structure of the system. Some of those considerations are reported below to show an example of them [Gall97]:

- The size of the system is increasing linearly.
- Between releases 2.00 and 5.00 and in release 7.00 some major activities can be observed.
- In general the changing rate of the system as a whole decreases.
- In the last examined release only few new *programs* were added.
- The structure of the system as a whole has become stable.

More detailed examinations discovered a particular sub-system which shows an anomalous behavior. Its highest growing and changing rate suggested to be a candidate for restructuring or even reengineering activities.

These considerations and others not reported here suggested to the authors that this method could really help in finding structural problems in a large software system.

The software release history is contained in a database as described in Section 3.4. Due to the huge size of the database a manual examination using the Navigator is a labor intensive and boring job (this tool is described in Section 5.1.1). The examination has been achieved setting up ad hoc programs written in C++. These programs are used to make the specific queries to the database. The tool Navigator has been used to gain additional information and to verify theories.

Chapter 5

STATE OF THE ART

The examination of an existing software system is a step of the software release history analysis. As described in Section 4.2.2, such examination should lead to make observations on the structure of the system. This section describes two tools which are useful during the examination phase of large software systems. The first part is an overview of the tools. The second one is organized by ideas and describes their advantages and defects.

5.1 Overview

5.1.1 Navigator

Navigator is a tool for visualizing software models. The software model must be specified in a database in terms of: *objects*, *relations* and *properties*. The tool is able to visualize graphically these three information about the software. For example, objects can be the modules of a software system, relations can be the relations among modules, and properties can be name or version number of the modules. A graphical layout is used to show these information together.

The primary purpose is to serve as a reverse engineering tool. It provides the ability to view various aspects of the software model in a graphical form, including the block structure, usage dependencies, versioning dependencies, history information.

The graphical layout contains two basic elements: boxes and lines. A box is used to visualize an object and its properties. A line is used to visualize a relation between objects, the name over the line is the name of the relation. Each object can be expanded to show its relations with other objects or can be collapsed to hide its relations and its related objects. The graphical elements have context-sensitive menus whose purpose is to show the properties of the related object or relation and to allow navigation capabilities. The navigation can start from any objects. A ROOT object is defined and it contains all the other

objects. The user can navigate through the structure expanding and collapsing objects. In this way he/she can filter the visualized information.

Figure 5-1 shows an example of visualization. The example is focused on the problem of examining software release history and uses the same data of Figure 4-1. Objects are the modules of the structure, the property is only the version number and the relations among modules depend on the system's decomposition. The ROOT object is the starting element. Expanding it, its related objects are shown: the software system at release 1.0 and 2.0. These two objects are decomposed by the relation "sub-system" in their sub-systems. In Figure 5-1 only the release 1.0 is expanded. The sub-systems A, B and C are not expanded. Expanding them would lead to visualize the module level of the structure. As shown in Figure 5-2.

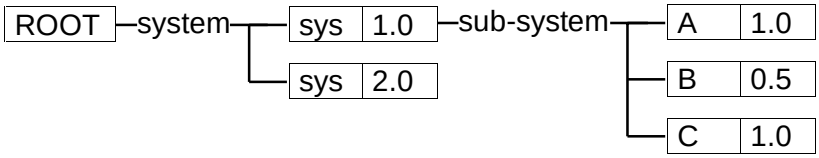

Figure 5-1 Example of use of Navigator.

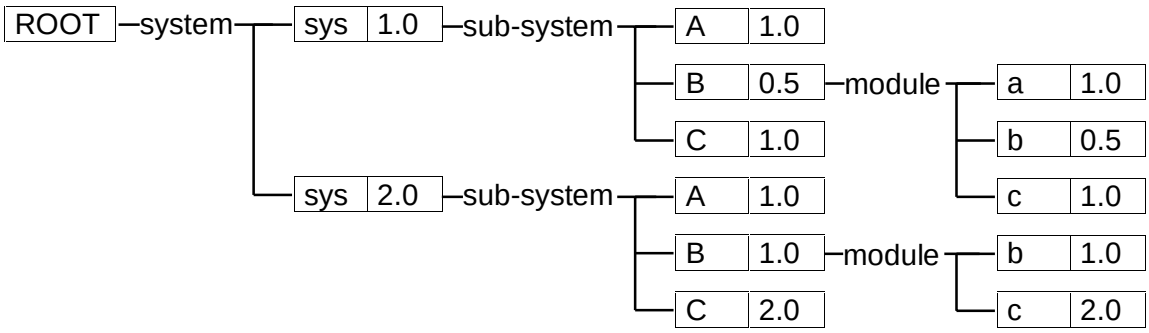

Figure 5-2 Example of use of Navigator.

Figure 5-2 shows another example of navigation with Navigator. The system at release 2.0 is expanded, and also the module level of the sub-system B. Such a visualization can be used to compare the two different releases of the system. It shows how the version numbers have changed and that the module a has been removed from sub-system B in the release 2.0.

Search mechanisms are implemented using event-driven menus. They allow to search for objects with specific properties. The objects found are displayed graphically. Navigation is possible from any objects using context-sensitive menus.

5.1.2 SeeSys Tool

SeeSys is a tool for visualizing software information. It was developed at AT&T by Marla J. Baker and Stephen G. Eick [Baker94]. The main purpose of

the tool is to visualize the statistics associated with code that is divided hierarchically into subsystems, directories and files. Its motivation arises by the problem of managing large software systems. In fact, the graphical technique developed is directly aimed at displaying large amount of information.

The method implemented by the tool makes the assumptions that the system can be decomposed in a hierarchical structure. The components of the decomposition are sub-systems, modules and files. It also assumes that some information about the components are available, such as software complexity metrics, number of changes number of programmers making modifications and number and types of bugs. The method provides a visualization of the components of the systems in terms of their structure and their information.

Figure 5-3 shows an example of visualization. The system is decomposed in three sub-systems (X, Y and Z). They are represented with the big boxes with harder edges. Each sub-system is composed of several directories. They are represented by the boxes contained by a sub-system box. The size of the boxes (sub-systems and directories) represents the relative size of the component in the whole system. Additional information is displayed by vertically filling the boxes. For example, the fill might represent the amount of bugs, or the complexity metrics or other information about the component.

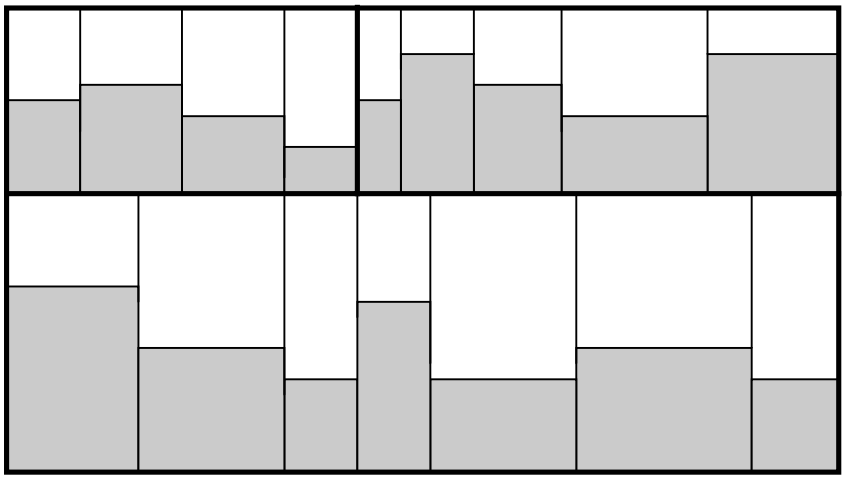

Figure 5-3 Example of visualization with SeeSys.

This approach represents the structure and the information about components in a concise form. Other functionality are:

- Interactive user interface. The user can retrieve the information about the displayed components using a mouse. Visualized statistic can be easily changed using a mouse.

- Zooming. The user can zoom in a particular zone of the graph to see the details that could be less visible because of dominant components. This problem happens when using area to visualize statistics:

components whose statistic is large are dominant to the detriment of small ones.

- Color is used to redundantly encode the statistic information in the graphs. Color doesn't take additional information.

5.2 Features of the tools

The previous section is a short description of the functionality of the two tools. This section focuses on their features to analyze how they can be useful for the software release history analysis. It's an analysis from the point of view of visualizing release histories. The purpose is to find the most interesting and original ideas that can help in the examination. The tools are evaluated on the base of several parameters.

5.2.1 Graphical Layout

Graphical layout concerns how the information is displayed in a graphical form.

Structure Visualization

The decomposition of a software system produces a set of components and relationships between components. Navigator displays this information using hierarchical 2-D charts made with boxes and lines that is the usual representation that programmers have of software systems. It is the standard and familiar way of representing the software structure. Relationships among components are clearly displayed. SeeSys uses rectangles for displaying components. Only two levels of the structure can be visualized: a component and its sub-components. The area occupied by sub-components is related to the relative size in the father's component. The focus is mainly on showing a property (i.e. the relative size) than relationships in the system structure.

Statistic Visualization

One component is the abstraction of a piece of code in the system. Several properties can be extracted from the code and assigned to the component: lines of code (LOC), number of bugs, complexity measures, version number and so on. Navigator shows only version numbers, other properties can be accessed opening a property window but they are not directly visualized. SeeSys was precisely developed to satisfy the need of visualizing statistics. Boxes are filled according to a property selected by the user. The fill provides an effective visual representation of percentages enabling quick comparisons.

Color

Navigator doesn’t provide any color capability for visualizing statistics. In SeeSys color is used to add redundancy information to the filling. Color doesn’t take additional information.

Use of available screen space

The screen space is the space that tool can use for its visualization. Both Navigator and SeeSys use a two dimensional layout. Navigator displays the hierarchical structure of the system. Often this visualization comport a waste of space. For instance, the visualization of a tree wastes all the complementary space not used by the tree itself. SeeSys uses all the available space because a standard layout is used for all the representations.

Structure Navigation

Structure navigation is the ability of moving among the elements of the system. Navigator was developed to satisfy this purpose. Starting from an object the user can navigate to the other related objects in both the directions (to higher or lower level of abstraction). Expanding and collapsing the relationships to the other objects is the main navigating functionality. A graph produced by SeeSys visualizes only two levels of abstraction, therefore navigation is limited to those two levels. Both the tools provides multiple windows, it means that the user can simulate the navigation creating several graphs.

5.2.2 User interface

User interface concerns how the tool interacts with the user. The user sees a program through its user interface.

Interactivity

Both the tools use an interactive interface. Navigator uses multiple windows, menus and mouse operations. SeeSys uses graphical buttons, a selector for the statistics and mouse operations.

Use of Mouse

In Navigator the mouse is used for navigating the structure, to open the property windows and to interact with the menus. In SeeSys the mouse is used to retrieve data from the graphical representation such as values, names or percentages, to make selections of values or components and in general for navigation aids.

5.2.3 Visualization of large information

Visualization of large information is a problem addressed in the present work. This section tests the fitness of the tools for large-scale software.

Global view

The global view concerns the possibility of displaying all the information about the system or sub-systems in one view. Generic observations can be extracted by such global view. Global doesn't mean at an high abstract level but refers to a view with elements of both high and low levels of abstraction. Navigator supports such feature with the zoom capability. When displaying both high and low level components of a large software system, the picture can be too big to fit into the screen. Zoom often leads to graphs that are unreadable. SeeSys provides an auto-scale capability. In fact, the graph is scaled into a fixed layout and can be visualized entirely. A limitation is that SeeSys can visualize only two levels of the system's structure.

Local view

As opposed to the global view, local view is the possibility of focusing on the details of one level of abstraction (higher and lower level as well). Both the tools allow this feature.

Selection

It's the possibility of displaying or highlighting some elements of the structure which have some common properties chosen by the user. Navigator provides a mechanism that allows to search for objects. The type and the properties of searched object is specified using a dialog-box. After the query is performed, the found objects are displayed in the graphical layout of Navigator. SeeSys doesn't provide a specific search mechanism. The user can select all the elements which have the same property value using the mouse and a slider bar.

Zoom

Zooming allows the user to focus on the details of a graph. Both the tools allows this functionality. Navigator provides zoom in and zoom out capability. SeeSys allows to zoom in the details of the main graph shown in Figure 5-3.

5.2.4 History Visualization

Different releases

Both the tools provide a functionality for displaying multiple releases. In Navigator the graph layout allows to display directly multiple releases as in Figure 5-2. SeeSys implements this feature with animation. The graphs of

successive releases are shown in succession like in a film. The user can see the evolution of the system looking at how the area changes during the time.

Comparisons among different releases

Navigator displays different releases into the same graph. In this way comparisons can be done on the same picture. When looking at many release of large system the graph may be to big to fit into the screen and zooming it becomes unreadable. SeeSys doesn’t provide comparisons on the same graph because of the animation. Comparisons are made on a perceptive level through the time.

Chapter 6

PROBLEM STATEMENT

This chapter presents the problem addressed in this Master's Thesis. The context of the problem is explained in Section 6.1. Then Section 6.2 is a concise description of the main question that this thesis tackles. Motivations for solving the problem and purposes are presented respectively in Sections 6.3 and 6.4. Section 6.5 discusses the advantages of the solution that is adopted in Section 6.2.

6.1 Problem context

The Software Release History Analysis is explained in Chapter 4. In this section the method is revisited in a generalized form in order to clarify its fundamental rules and principles. The description is a support for the next Chapter 7 where the requirements for the visual representation are explained and implemented. The generic approach used in the description allows to identify a visual representation that is adaptable for future development of the method.

6.1.1 Approach for system abstraction

Understanding a large software system by examining its source code is an arduous task. The amount of details present in the source code would overwhelm the developers. To simplify the human comprehension level abstract view of the source code are used: software engineers use high levels of abstraction to navigate through the numerous software components and their interconnections and statistical analysis is also used to extract the essential information from the code.

The Software Release History Analysis uses an abstract representation of the system that is generated in two steps:

- System decomposition: the system is decomposed in two elements: components and interconnections. Each component is associated to a part of the system source code.

- Measuring software attributes: for each component of the decomposition a set of attributes can be measured. The values of attributes are directly calculated on the associated source code.

The result of the decomposition (i.e., components and interconnections) is called *structure* of the system. The attributes are the properties of the components of the structure. The two steps are explained below.

System decomposition

The structure of the system is defined in terms of *modules* and *relationships* between them.

Modules

A module is the basic software component of the decomposition. It has an internal structure that is invisible to the other modules and an external interface which exports its functionality. It can eventually be decomposed in other modules or directly be implemented in software. The main concern of this abstraction is reducing the complexity and moving to a more abstract level than code. According to [Lehman85] and [Turski96] when dealing with large systems, system evolution is driven more by changes in functionality than by low-level tinkering with code. Changes are reflected by added, removed or handled modules. Therefore, evaluation should be based on module more than on code. Considerations about this behavior are discussed in Section 2.4.

This is also consistent with the infrastructure trends of telecommunication companies described in [Linden95]. The trend is to use a modular structuring which encapsulates a feature in a module instead of distributing its functionality over the system. This modular structure makes it easier to add, remove and manage the functionality of the system.

The case study considered in this thesis has the layered architecture that is described in [Shaw89]. The system is organized hierarchically with each layer providing service to the layer above. Lower layer hides the implementation to the higher ones.

Relationships between modules

Relationships between modules are related to the method used to decompose the system. The structure obtained by the decomposition depends on the method used: "part of" decomposition produces a hierarchical structure, "use" decomposition produces more complicated graphs. The case study has a layered architecture, so modules can import only blocks from lower layers. This decomposition produces a hierarchical structure.

Measuring software properties

The previous step extracts the structural information from the source code. Other useful information can be extracted. The theory of software measurement

described in Section 2.5 addresses such a problem of mapping software code to a set of attributes. The objective is to create a concise and abstract representation of a piece of software in terms of a set of attributes. The attributes can embody some characteristics of quality or software property that are under investigation. For example, considering a piece of code we can calculate the size in terms of lines of code, complexity using complexity measures, age in terms of version numbers, error proneness in terms of bugs detected and other measures [Fenton96].

Summarizing the two steps, the code of the system is divided in parts and each part is linked to an element of the structure i.e. a module. On each piece of code a set of attributes can be measured. These attributes are a short representation of the code and can be associated to the module. In this way each module of the structure has a set of properties whose values are the values of the attributes measured on its software code.

This process of abstraction is exemplified in Figure 6-1. The abstract representation constitutes of three modules interconnected according to a decomposition model. Each module has three properties: the name, the number of lines of code and the version of the module.

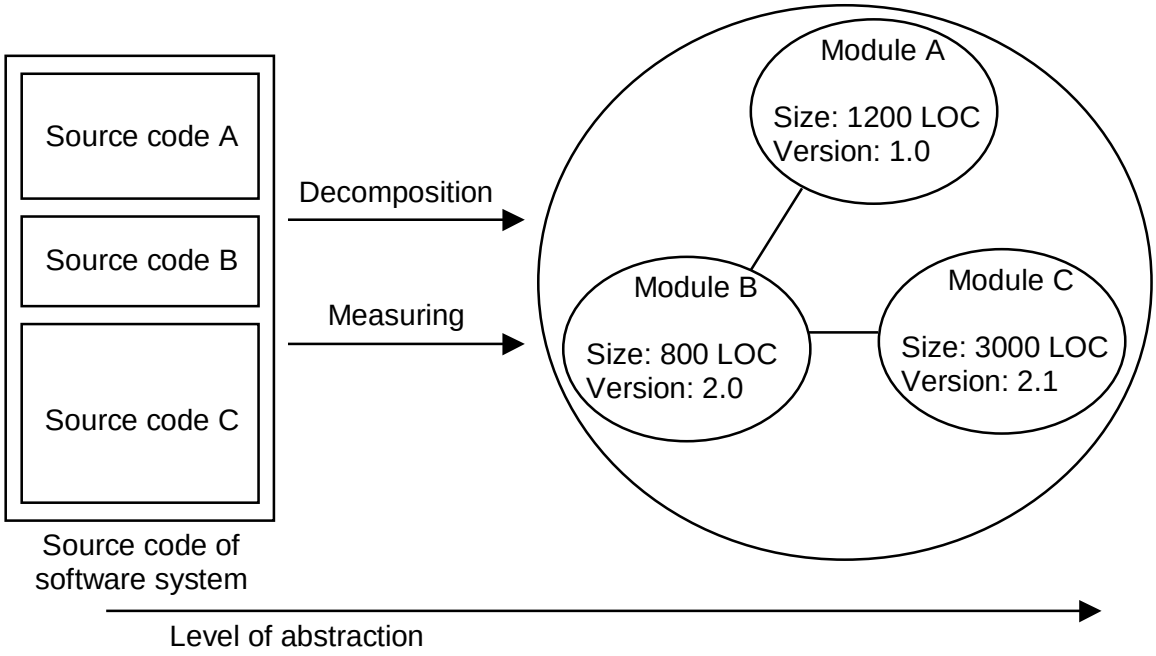

Figure 6-1 The abstraction process.

6.1.2 The Software Release History

The innovative approach of the Software Release History Analysis is the use of the history of the system to evaluate it. The historical evolution of a software system is signed by release deliveries. As described in Section 2.2.2 each new release incorporates changes to the code of the system. Software Release History tracks this evolution.

According to the abstract model developed in the previous Section 6.1.1 system evolution can be detected by modifications of two elements: structure and software attributes. The system structure is modified when modules are added to or removed from the system. The software attributes can change when the code that has been used to measure them is modified. For example, a new module is added to implement a new functionality, in this case the new structure is obtained from the old one plus the new module. Another example, the code of a module is re-written, in this case the module's version number increases. Therefore the abstract representation modifies its information as the system evolves. For each system release an abstract representation can be extracted from the source code.

The Software Release History captures the evolution of the abstract representation. It is constituted of three entities:

- Time: this coordinate is expressed in *release sequence number*, RSN as defined in Section 4.2.1. The advantages are that the system is precisely defined at times of releases and that the release intervals correspond to well-defined units in the system life-history.

- Structure: the system is decomposed in modules as described in the previous section.

- Attributes: a set of attributes, such as version number, size, complexity, detected bugs are measured on the modules.

The time line is discrete and a value of it identifies an unique release of the system. For each element of the time line the abstract representation (structure and values of attributes) is extracted from the code. The information is stored in a database. The database is called the *Software Release Database*.

In Section 4.2.1 this approach is described as it was first proposed by [Gall96] and [Gursch96]. Only one attribute is considered that is the version number of the modules. In this section the method has been generalized to any attribute that can be measured on code.

6.1.3 The examination process

The Software Release History Analysis is a scientific methodology for evaluating the structure of a system. The system is examined to produce some observations. Observations are made on something that we think is wrong, anomalous or has a strange behavior. For example, a component of the system which always changes its implementation. Observations arise when comparing the real system (i.e., the system being analyzed) with the ideal system which embodies the maximum quality. The differences between the real system and the ideal one are the observations. They are the points in which the real system doesn't meet the characteristics of quality that we describe in the ideal system. The ideal system depends on the level of quality that we require or on the level of examination we want to do (superficial examination or more detailed).

Section 4.2.2 describes an example of the examination process used by Gurschler [Gursch96] to extract observations from the Software Release History. He adopted a cognitive approach to the problem. The first step is to write a list of queries which represents the parameters of quality. The ideal system is the system which has the maximum quality for all the queries in the list. In the second step the real system is submitted to the queries. The answers describe the real system in the terms of the parameters chosen in the first step. The comparisons between the answers and the ideal ones allow to identify the outliners. Outliners are the parameters in which the real system doesn't match the ideal one that is what we would like to have. The outliners have a behavior that is not accepted by the quality level of the ideal system. Outliners are the sources of observations.

6.2 Problem description

The Software Release History Analysis is a methodology for evaluating an existing system in order to formulate considerations about its structure or evolution of its structure. The purpose is to increase the comprehension about the system.

The considerations are formulated examining the Software Release History as described in Section 6.1. The volume of data contained in the Software Release History causes to emerge a problem: how can useful information that is hidden within the data be extracted ? The ability to extract and to understand this information is the key to increase the comprehensibility of the system. Section 6.1.3 describes the issues of examining and formulating observations about the system. In the previous work [Gursch96] Gurschler adopted a cognitive approach as described in Section 6.1.3. The present work proposes a perceptive approach to the problem.

Visualization is an emerging technology for understanding large and complex data sets. The main objective of the present work is to investigate the possibility of visualizing the information contained in the Software Release History. The aim is to help the task of examining the Software Release History. It is the intent of this thesis to show that a visual representation of data can help to extract useful information. It is not an intent of this thesis to interpret the extracted information. Motivations and purpose of this choice are explained below.

6.3 Motivations

This section discusses the reasons for solving the problem just stated.

Valuable method for structural assessment

The effectiveness of the Software Release History Analysis has been proved in the work of A. Gurschler [Gursch96]. The method has been applied to the case study described in Chapter 3. Evaluations produced useful considerations about the structure of the system. These considerations are valuable information for software engineers to identify potential shortcomings or problems in their system.

Shortcomings of current approaches

The examination of the Software Release History requires to extract the information from a database. To extract the needed information Gurschler wrote some programs. The system is too large for a manual examination using the Navigator. In fact, the Navigator doesn't provide a way to extract statistics about the system. For this reason he decided to query the data directly from the database. The choice of Gurschler points out the shortcomings of the Navigator for supporting the Software Release History Analysis.

Chapter 5 describes two existing tools that could be useful for examining release histories instead of direct use of the database. Both the tools are explained in Section 5.2 in terms of their features for supporting the examination of release histories. The most interesting and original ideas have been highlighted and are summarized below:

- Visualization of structure (Navigator and SeeSys)
- Navigation of structure (Navigator)
- Visualization of statistics using filling (SeeSys)
- Interactivity and use of mouse (Navigator and SeeSys)
- Auto scaling of graphs (SeeSys)
- Zooming (Navigator and SeeSys)
- Visualization of multiple releases (Navigator and SeeSys)

These ideas could be useful for examination the Software Release History but the existing tools don't provide an adequate approach.

Navigator is mainly focused on displaying the structure of the system and providing a navigation capability. The shortcomings are:

- Other properties except version numbers are not directly shown but are accessible via menus. They are useful when interpreting the behavior of blocks [Gall96].

- The graphs of large systems are too big to fit into the screen. Zooming leads to unreadable graphs.

- The comparisons of structural elements or elements of different releases are difficult for large systems. Navigator doesn't provide this task directly. The main problem is that often the elements, which should be compared, are far from each other.

SeeSys was developed to visualize the statistics of a large software system. This was the primary objective. The critics are:

- The visualization of the structure is limited to two levels of abstraction.

- The elements of the structure are shown using rectangles whose dimensions are related to the relative sizes of the elements in the system. This could not be the most interesting property when examining the structure of the system and could bias the visualized statistic.

6.4 Purpose

The purpose of this thesis is to develop a visual representation to examine the history of the release software. Such a representation is an aid to the Software Release History Analysis. The objectives that the thesis want to achieve are:

Incorporating the advantages of visualization

The works of S. Eick and D. Fyock show that visualization is a promising solution for understanding large and complex data sets [Eick96] [Eick96a]. Traditional approaches for analyzing data, such as spreadsheets, ad hoc queries, statistical analysis are unable to handle the volumes of data that businesses want to analyze. Visualizing the information offers several advantages that are discussed in the next Section 6.5.

The main purpose of this thesis is to develop a visualization technique for the Software Release History. The objective is to take the advantages of visualization for improving the examination process of the Software Release History Analysis. The requirements for improving the examination process are investigated in Sections 6.1and 7.1.

The use of Color and Third Dimension

The visualization technique presented in this thesis focuses on the use of two visual attributes: color and third dimension.

Color can encode additional data in traditional graphs or can be used to add redundant information in traditional graphs. The main advantage of color is that it stimulates the human visual system. In fact, color processing is an

independent perceptual process in the human vision system and its use doesn't overload the comprehension activity. In this way colors can provide more information in the same graph. Moreover they enable the humans' pattern matching skills. Colorful displays also engage the user and are visually interesting and pleasing. In the present thesis the intent is to use the color to take the advantages of this vision: encoding additional information, pattern detection and visual pleasure.

For software data visualization the third dimension is a new and under examination area. The main advantage is that in a natural way more data can be packed onto the screen without overloading it. The information content (the ratio of information to pixels) can be increased by a factor of 10 [Tufte83]. It's difficult, however to find 3-D representations of shapeless data like software information. In this thesis the third dimension is used for two intents. The former is to give a 3-D fashion to traditional hierarchical representations and the latter is to visualize the historical information.

6.5 Advantages of Visualization

The advantages of visualization are numerous [Tufte83], [Eick96]. This section presents the mostly mentioned in literature.

Cognitive to perceptive

The process of understanding abstract information is a cognitive task. Visual representations are an aid to humans' comprehension and understanding. For example, a logical hierarchy itself is not a tree. The tree shape is a visual representation of the hierarchy that helps our understandability. Visualization shifts the understanding process from being a cognitive task to being a perception task.

Pattern Detection

Visualization presents the information in a pictorial form that enables the humans' pattern matching skills. Humans have developed a sophisticated visual technique that allow to detect patterns. Through visualization it is possible to adopt this technique to detect patterns within sets of data. S. Eick says about visualization: "It links the two most powerful information processing systems - the human mind and the modern computing." [Eick96].

Effectiveness

Examining huge textual tables is a boring and time expensive job. Visual representations are an alternative to textual tables. In fact charts are more effective in creating interest and in catching the viewer's attention. Visual

relationships are more easily grasped and remembered. They can also encourage the eye to compare different sets of data simultaneously.

Chapter 7

VISUALIZING THE SOFTWARE RELEASE HISTORY

This chapter presents the approach that has been developed to solve the problem stated in Chapter 6. At first the requirements for the visual representation are detected and motivated in Section 7.1. Then in Section 7.2 the adopted approach is described. Section 7.3 shows how the visual representation is an aid for examining the Software Release History. In Section 7.4 the advantages of the approach are discussed.

7.1 Requirements for visual representation

The examination process is described in Section 6.1.3. Its task is to identify outliners or to locate unusual behaviors in the system. This section describes the features and requirements that the visual representation must support for examining the Software Release History.

Visualization of structure, attributes and history

The abstract representation described in Section 6.1.1 contains two entities: structure and attributes. History is the third element described in Section 6.1.2. To examine the Software Release History all of them have to be considered. The visual representation must support their visualization.

Support for large volume of data

The modular decomposition of large software systems, on the order of millions lines of code, can contain thousands of modules. The case study examined in this thesis consists of 13 millions of lines of code and 2300 *programs* (in the last release). The visual representations must support such a large amount of data and have to reduce the visual complexity of such a huge set of data.

Global and local representation of data

Visualization must support both, global and local representation of data. The former is the possibility of displaying simultaneously all the entities (structure, attributes and history) in one view. Observations about the global behavior of the system can be extracted by such a view. The latter is the possibility of focusing on the details. These parameters have been defined in Section 5.2.3.

Support of comparisons

Examination requires the capability of comparing the data of Software Release History. Three types of comparisons are required:

- Comparisons among modules to evaluate their differences and their relationships. For example, the comparison of all the modules belonging to the same sub-system at one system release can reveal outliner modules.

- Comparisons of two or more different attributes to evaluate the dependencies between them. For example, comparing the size of modules to their complexity measure we could induce that large modules also have high values of complexity.

- Comparisons of different system releases. This is the innovative approach of using the Software Release History. For example, comparisons between different releases could lead to identify the behavior of a module which is always changing its version number. This could be a symptom of is bad implementation.

Navigation

The large nature of considered systems suggests that a capability for navigating should be supported. Navigation allows to move through the data for changing the viewpoint of the representation. Different points of view are necessary to understand the visualized data better.

7.2 Approach

7.2.1 Overview

The Software Release History described in Section 6.1.2 is constituted of three entities: time, structure of the system, measures of attributes. These entities are visualized using one three-dimension diagram (3-D diagram). The coordinates are called *x*, *y*, and *z*. In a 3-D diagram the entities are displayed in this way:

- Time: the coordinate *z* stores the time information. This time coordinate is expressed in *release sequence number* (RSN).
- Structure: for each RSN the system has an associated structure. The structure is displayed using 2-D or 3-D graphs and it's spatially positioned along the coordinate *z* at the value of its own RSN.
- Attributes: each diagram can display one attribute at a time. Each structure is identified by the RSN and has its own set of attribute values. These values are shown using colors. The values are mapped to a color scale and so each color of the diagram represents a value.

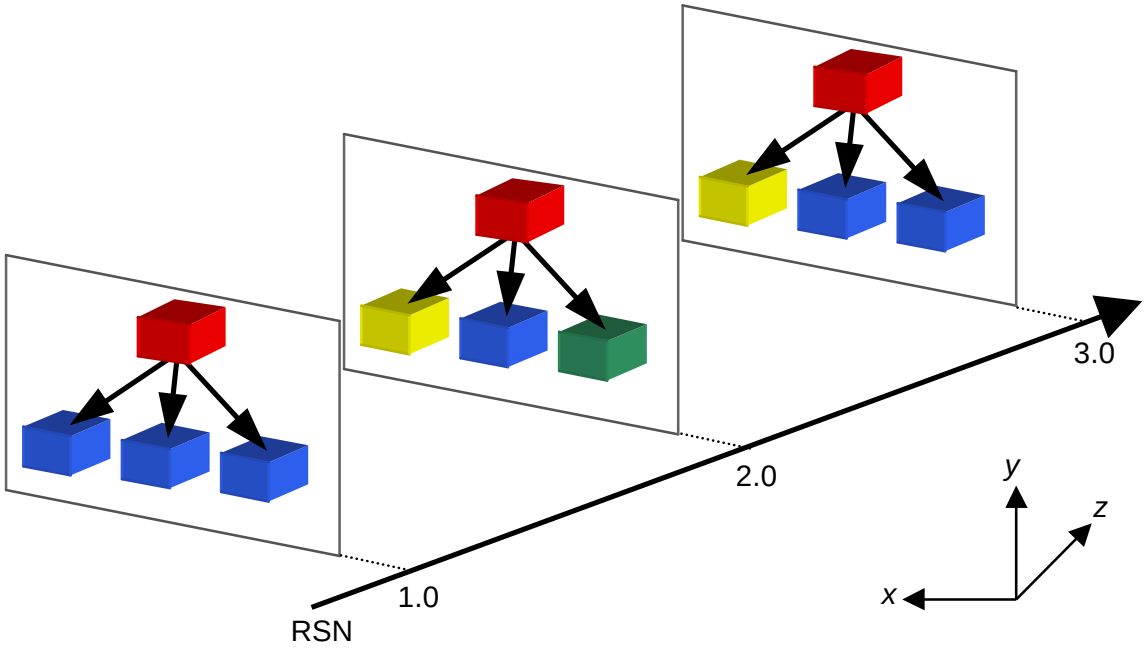

Figure 7-1 A 3-D diagram.

Figure 7-1 shows a graphical representation of this approach. For each system release (1.0, 2.0 and 3.0) a graph shows the structure of the system. The graphs can be of two types, 2-D or 3-D. In figure a tree structure is used as an example. Each colored block is associated to a module of the abstract decomposition. The colors are used to visualize a software attribute. Each color is mapped to an attribute value through a color scale. For example, Figure 7-1 is the visualization of the values reported in Table 7-1 where the attribute is the size of modules. Three system releases are considered. In all the releases the system has the same structure constituted of three modules (A, B, C). Blue color represents a size between 0 and 1000 LOC, green a size between 1000 and 2000 LOC and yellow a size bigger than 2000.

RSN	1.0			2.0			3.0		
Module	A	B	C	A	B	C	A	B	C
Attribute (size in LOC)	300	400	500	1200	400	3000	800	500	3000
Color									

Table 7-1 An example of data for Figure 7-1.

This visual approach provides an immediate representation of the data. Using the color scale it's possible to relate the colors to the values, and so immediate comparisons can be done. The advantages of this approach are reported in Section 7.4. The graphs for displaying the structure and the use of color are described in the next two sections.

7.2.2 Structure visualization

Section 6.1.1 describes how the system structure is abstracted from the source code. The system is decomposed in two elements: modules and relationships. The decomposition rule determines the type of structure. For example, the case study has a layered architecture and its structure is a tree hierarchy, represented in Figure 3-2. The elements of the decomposition are mapped to graphical objects. The composition of these graphical objects produces a graph which is the visualization of the system structure. The graphical elements and the graphs are explained below. The meaning of the colors is skipped in this description, they are explained in next Section 7.2.3.

Graphical elements

Modules and relationships are the elements of the structure. Four types of graphical objects can be identified:

- *System module*: it is the topmost module of the decomposition. It represents the whole system
- *Generic module*: it is a generic element of the system decomposition.
- *Relationship*: it is the relation that connects two modules.
- *Percentage bar*: it is a reduced representation that allows to visualize a set of modules.

The four elements are described more detailed below.

System module

The system module represents the whole system. It is visualized with a colored 3-D sphere shown in Figure 7-2. Its color depends on the attribute values of the module. It is the visual representation of the whole system from which the decomposition originates. For example, it's the topmost element of Figure 3-2 of the case study, i.e. the *system* element.

Figure 7-2 The system module.

Generic module

The generic module represents a module of the system decomposition. It is visualized with a colored 2-D cube shown in Figure 7-3. The color depends on the attributes of the module. For example, considering Figure 3-2, it is the visual representation of a *subsystem* or a *module* or a *program*, i.e. any module produced by the decomposition.

Figure 7-3 A generic module.

Relationship

A relationships exists between two modules. Its graphical element is a line, the color doesn't have any meaning. The object is shown in Figure 7-4. It is used to connect two graphical elements, for example two modules.

Figure 7-4 A relationship.

Percentage Bar

The percentage bar is a reduced representation of a set of modules. It allows to visualize an attribute of the modules in term of percentages. The same attribute is considered for all the modules of the set. The attribute values are used to calculate the percentages of blocks which have the same value. For example, we can consider 6 modules and their version numbers as attribute; 3 modules have version number 1.0, 2 modules version number 2.0 and 1 module version number 3.0. The percentages are: 50% for version number 1.0, 33% for version number 2.0 and 17% for version number 3.0. The bar allows to visualize these percentages in a graphical form.

The bar is composed of a set of colored blocks. Each block has two properties: relative size and color. The relative size is proportional to the percentage of modules that have the same value of the attribute. The color is selected by the value of the attribute. For example, red is assigned to version number 1.0, green to version number 2.0 and blue to version number 3.0. The Table 7-2 shows the example just described.

Modules	A	B	C	D	E	F
Version number	1.0	1.0	1.0	2.0	2.0	3.0
Percentage	50%			33%		17%
Bar						

Table 7-2 Example of calculating percentages.

The graphical object is shown in Figure 7-5. It is helpful for the visualization of the attributes of hundreds of modules. The percentages and their associated values are perceptively grasped. For quantitative comparisons size is the most effective perceptual data encoding variable [Eick97].

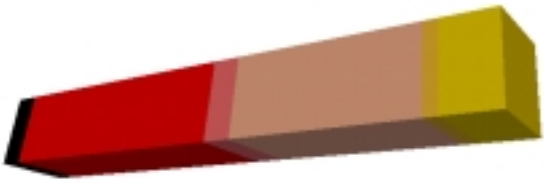

Figure 7-5 The percentage bar.

Graphs

Graphs are obtained composing the graphical objects described above. They are used to visualize the structure of the system in a graphical fashion. Using 3-D diagrams both 2-D and 3-D graphs can be displayed. Two main issues have been identified when dealing with the three dimension display:

- how to arrange spatially the graphical objects ?
- how to support large volumes of data ?

The solutions to adopt depends on the system structure and on the amount of data that has to be visualized. The nature of the case study forced to implement specific solutions for hierarchical structures. The techniques for visualizing this kind of structure are explained in Appendix B. This section reports only the results of the implementation.

A hierarchy of elements can be visualized with a tree structure. The tree structure is composed of father and children elements. Each father element is

connected to several children elements and can eventually be the child of another father. Fathers and children are mapped to the hierarchical elements so that the children represent the elements immediately below in the hierarchy to the element represented by the father. The topmost father represents the topmost element of the hierarchy. The tree can be visualized both with 2-D and 3-D graphics.

3-D Tree

Three dimensions are used to spatially distribute the objects of the tree. In this way more structure elements can be packed in one view..

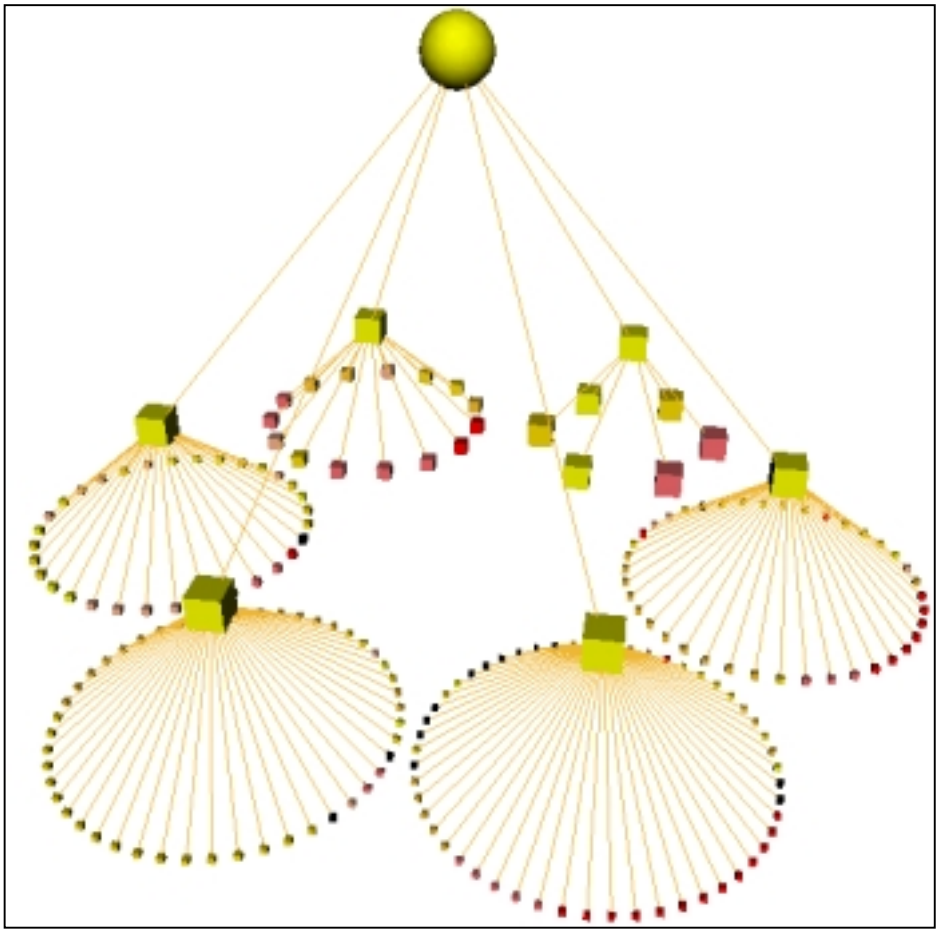

Figure 7-6 3-D representation of a system.

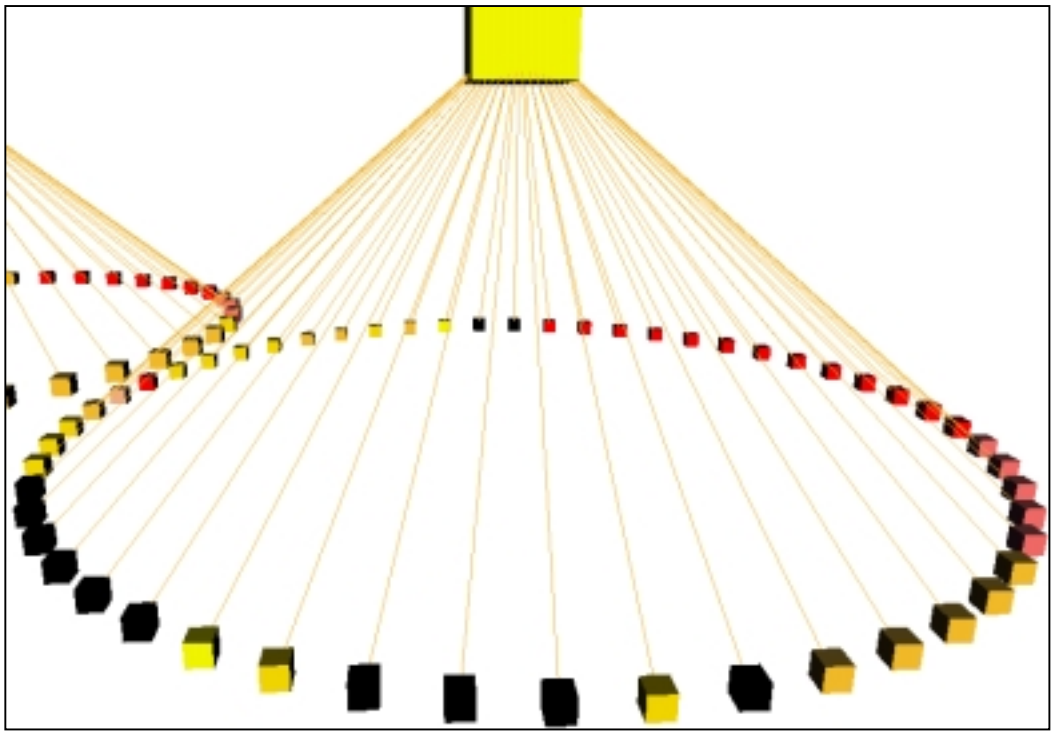

Figure 7-7 Zoom on a *subsystem.*

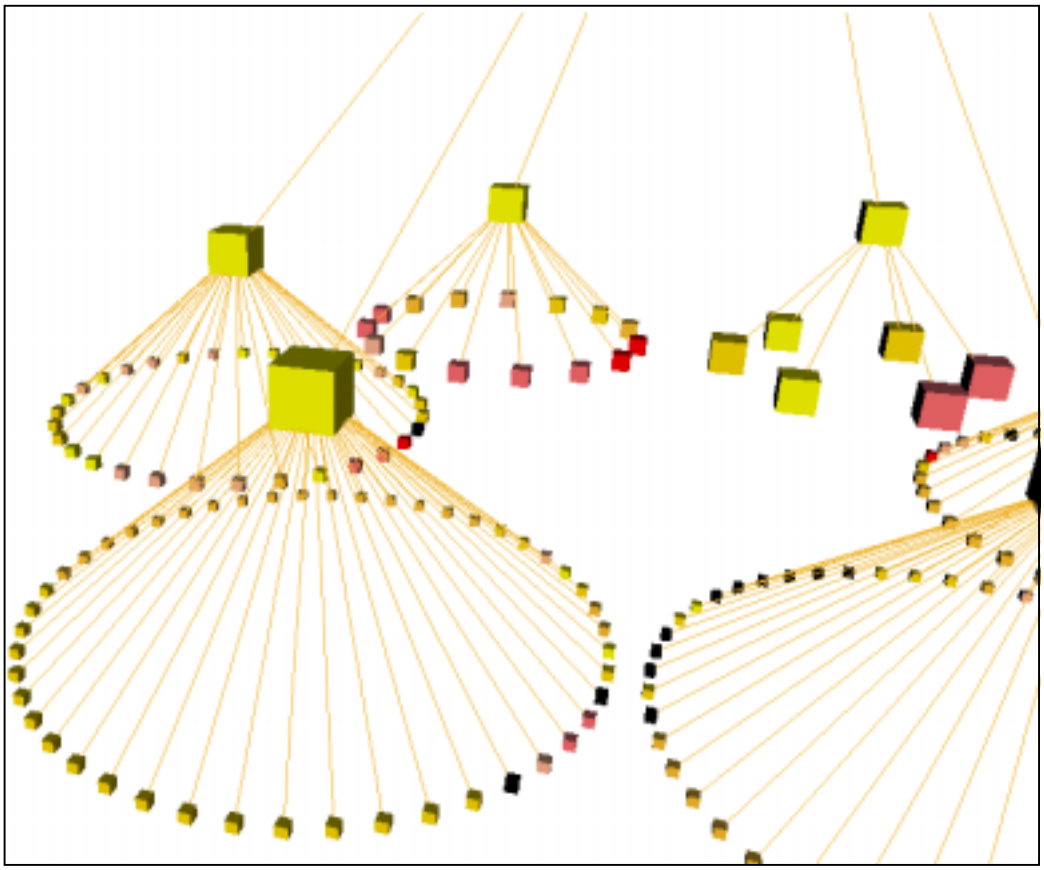

Figure 7-8 A generic view on the *subsystems.*

Examples of 3-D trees have been produced using the data of the case study. Figure 7-6 shows a hierarchical structure. The sphere at the top of the tree represents the system element. The second level of the hierarchy is constituted of *subsystems,* and at the third level the *modules* are located. The program level is not represented. The structure belongs at a generic release of the system. Figure 7-7 is a zoom on a *subsystem* to show its modules. The third and last picture, Figure 7-8, is a generic view through several *subsystems* and their children, i.e. the *modules*.

2-D Tree

Only two dimensions are used to display this type of graph. The third dimension is only used to give a 3-D form to the graphical objects. The main advantage of this representation is the possibility of bringing them near to each others in a compact form, this feature is shown in Section 7.2.4. Figure 7-9 shows an example taken from the case study. Figure 7-10 represents the same subsystems of Figure 7-6 in a different way. It shows the use of the reduced representation, i.e. the percentage bars.

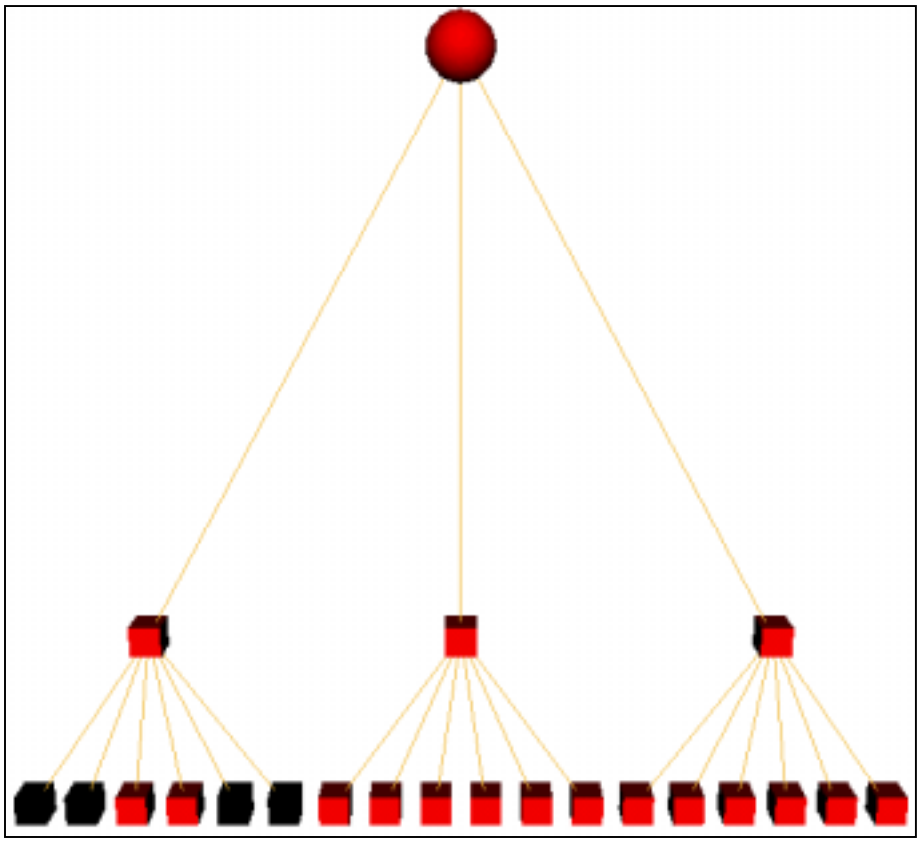

Figure 7-9 A 2-D tree of a system.

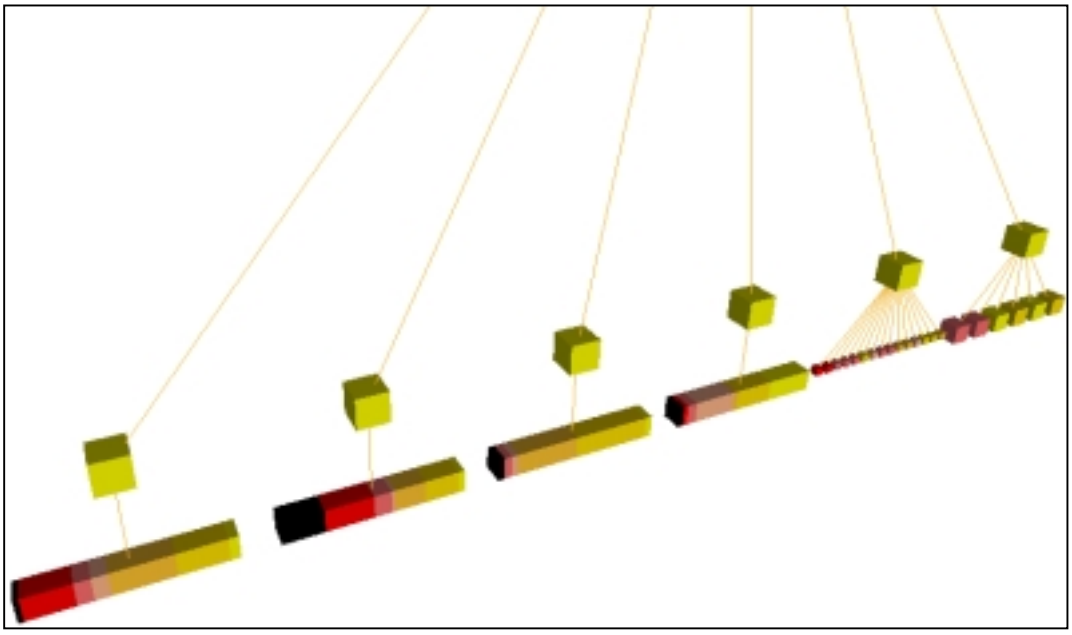

Figure 7-10 A zoom on the *subsystems* of Figure 7-6.

7.2.3 The use of color

Colors are used to increase the informative content of a diagram. Each module has associated a set of attributes, like version number, size, complexity measure, as described in Section 6.1.1. The values of an attribute can be mapped to a color scale in a way that each color represents a value or a set of values. Then each system module and the generic module are painted, according to the color associated to their attribute values. The coloring technique of the percentage bar is explained in Section 7.2.2. The numerical information is presented in a visual manner. The advantages are reported in Section 7.4.3. Each diagram contains a color scale and allows to visualize only one attribute at a time. An example of color scale is reported in Section 7.3.1.

7.2.4 History visualization

The graphs described in Section 7.2.2 allows to display only one system release at a time. The structure and the colors refer to the module's values of a specific system release. To add the information about the history, the 3-D diagram uses the third dimension. As explained in Section 7.2.1, different releases of the system are displayed placing their graphs along the *z* axis. The *z* coordinate is expressed in release sequence number. For each RSN the graph is created with the information of the release individuated by RSN. The obtained graph is shown. In this way the structure and the historical information is displayed in the diagram. In Section 7.2.2 two types of graphs have been described. Their use for history visualization produces two types of graphs. They are explained below.

Multiple 3-D Tree

Multiple 3-D Tree graphs are obtained placing several 3-D tree graphs along *z* axis. An example taken from the case study is shown in Figure 7-11.

Multiple 2-D Tree

Multiple 2-D Tree graphs are obtained placing several 2-D tree graphs along *z* axis. Figure 7-12 shows an example taken form the case study. Figure 7-13 is another example using the percentage bars.

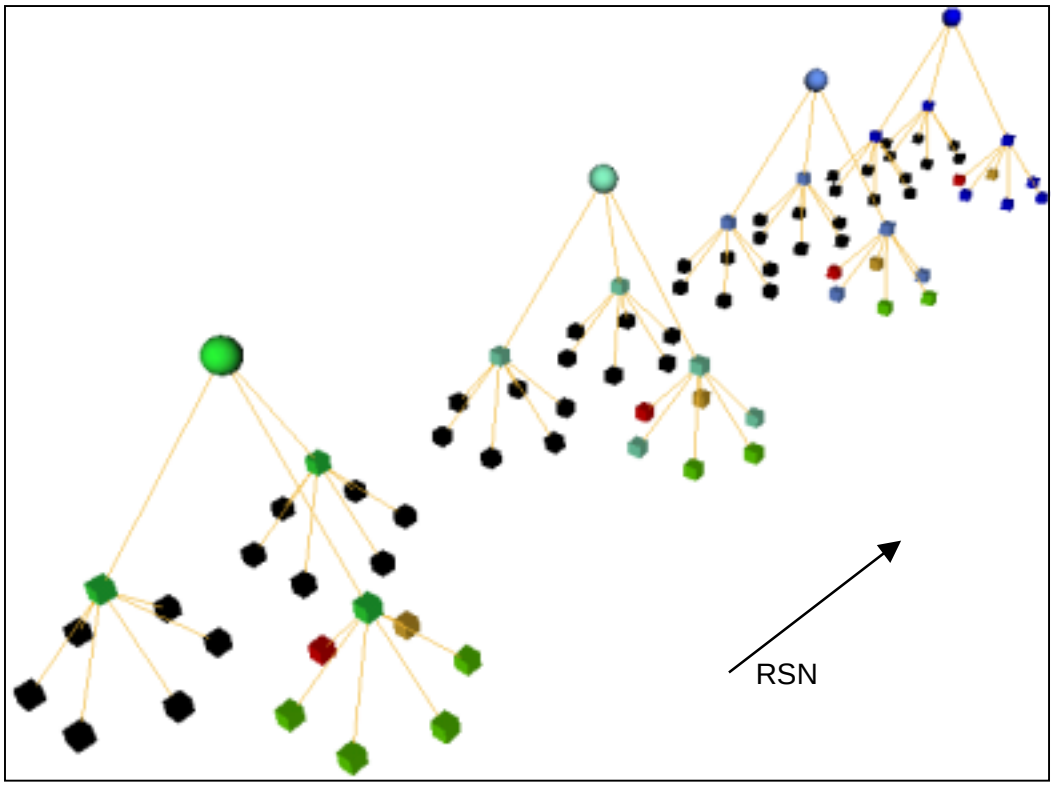

Figure 7-11 An example of a multiple 3-D tree.

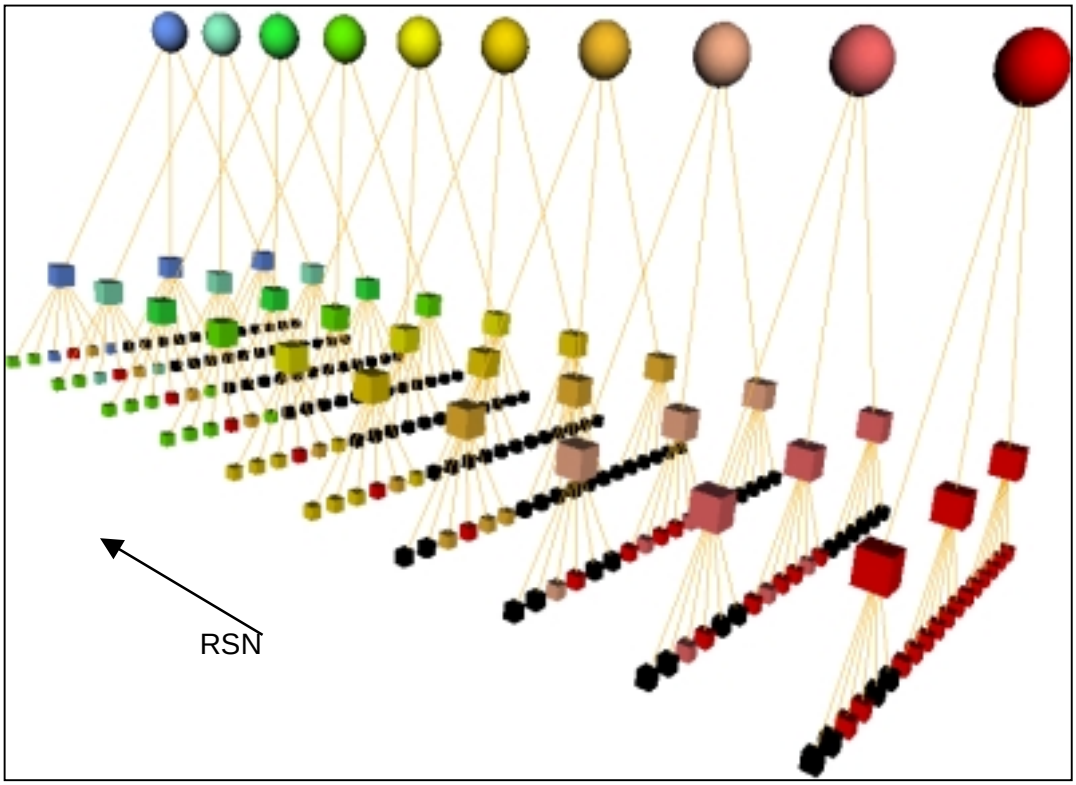

Figure 7-12 An example of a multiple 2-D tree.

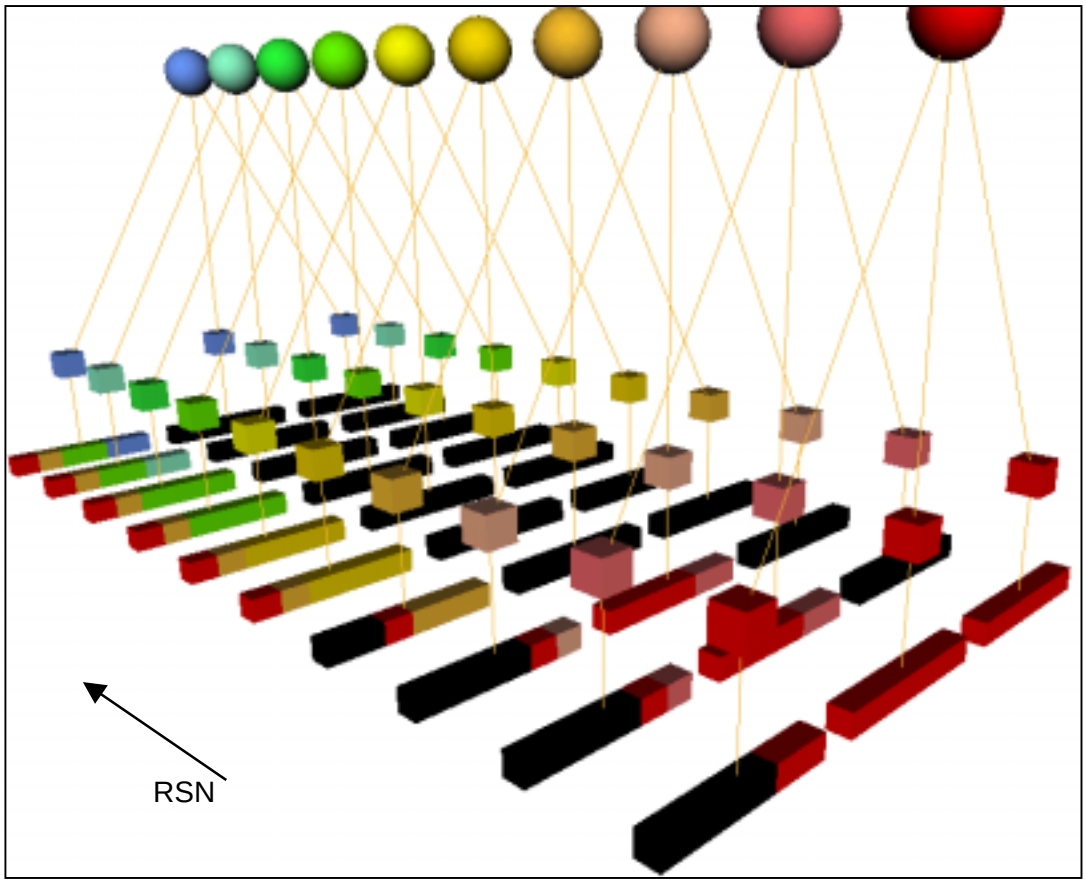

Figure 7-13 An example of a multiple 2-D tree with percentage bars

7.2.5 Navigation

3-D display allows to view the graphs from different viewpoints. The observer can choose the best view for looking at the data he/she is interested on. The possibility to change the layout of the graphs interactively is another feature that helps the navigation of the data. Several requirements have been identified and have been implemented in the visualization system described in Chapter 8. These features are reported here to complete the description of the adopted approach:

- Support for navigating in the 3-D diagram: The user can change his/her position and orientation in the 3-D space. The interactive display implements capability for moving, rotating, zooming and predefined automatic movements.

- Support for changing the layout of the graphs: The graphs have several properties such as size of objects, graph dimension and location. The user can set them up changing the parameters in a property window.

- Support for navigating through the structure elements and through the releases: The menus of the property window allow to select the modules and the system releases to show.

- Support for retrieving data from the diagrams: Using the mouse the user can get the attribute values of the modules and of the percentage bars.

- Support for multiple views: At the same time the user can open multiple views containing different types of visualization of the system and can arrange them to make comparisons among the visualized data.

7.3 Examining with Visualization

The purpose of this thesis is to develop a visual representation to help the examination of the Software Release History. The adopted approach has been presented in Section 7.2. This section shows how the visual representation can be used to examine the Software Release History. The main features and the ways of use are presented. The section also proves that the requirements stated in Section 7.1 have been satisfied. It is an intent of this section to show how to generate observations about the system structure. It is not the intention of this thesis to give an interpretation of the observations.

The Software Release History of the case study is used to give examples of visualization and to provide the data for examinations. The available database has the limitation that it contains only one attribute. The attribute is the version number of the modules of the structure, described in Chapter 3. The discussion is limited to this attribute. However, this doesn't diminish the general approach proposed in Chapter 6. In fact, the thesis's intent is to investigate how the use of visual representations is an aid to extract useful information from large amount of data, but it's not an intent to interpret the extracted information.

The data of the case study are not ready for a direct visualization. The elaboration to prepare them for visualizing leads to the Change Release Database presented in the next Section 7.3.1. The Change Release Database is suitable for the requirements of the Software Release History as described in Section 6.1.2. It contains the following three entities

- Time: the RSN is the time coordinate

- Structure: *system*, *subsystems*, *modules* and *programs* are the modules of the structure. Each element has an attribute, its version number.

- Attributes: the only attribute is the version number. For *system*, *subsystems* and *modules* it is the RSN. For *programs* it is the change sequence number (CSN).

7.3.1 The Change Release Database

The data provided by the case study are explained in Section 3.3. For each system release *program* elements have a number which is their version number. The version number is used to identify uniquely the implementation of

the element. The numeration of version numbers is independent for each module and it is independent of the system release numeration, so that it could happen that at system release 3.0 *program* A has version 2.0 and *program* B version 4.1 without being any numerical dependencies among the numbers. *Subsystems* and *modules* don't have any version number.

The data don't fit for the graphical representation proposed in this thesis because a common color scale cannot be set up. A new type of numeration must be assigned to the elements. The adopted approach uses a common numeration for all the elements of the structure. The numeration is based on the *release sequence number* (RSN) and on the *change sequence number* (CSN).

The RSN is an integer value used for numbering the system releases. Each system release has a value for identifying it. This value is called *system version*. The release sequence numbers are mapped to the system version values. The table shows the mapping for the case study.

System version	1	2	3	4	5	6	6.01	6.02	6.03	6.04
RSN	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***

System version	6.05	6.06	6.07	6.08	6.09	6.10	6.11	6.12	7	8
RSN	***11***	***12***	***13***	***14***	***15***	***16***	***17***	***18***	***19***	***20***

Table 7-3 Mapping from system version to RSN.

For each element of the structure (*system*, *subsystem*, *module* and *program*) the numeration is described below. This numeration is used in the Change Release Database.

System **level**

The structure of each system release contains a *system* element. The version number of this element is the RSN of the release to which it belongs. For example, according to Table 7-3 the *system* element of release 6.01 has version number 7.

Subsystem **and** ***Module*** **level**

In the case study *subsystems* and *modules* don't have any numeration because they are an abstract entity. Their version numbers are the same of the system element to which they belong. For example, if *subsystem* A belongs to the *system* with RSN 4 then A has version number 4.

Program **level**

The numeration of the *program* elements is based on the change sequence number, CSN. At a specific release the change sequence number of a *program* element is the RSN of the release where it had the latest change. An example is reported in Table 7-4. Two *programs* are displayed: *program* A and *program* B.

The first row shows the RSN, so that each column represents a system release; the second and fourth rows represent the versions numbers of A and B as they are in the case study. The third and fifth rows are CSN of A and B as they are used in the Change Release Database. The *program* A changes its version number at releases 1, 2 and 5, so its CSN is the sequence < 1 2 2 2 5 5 >. In releases 2, 3 and 4 the version number is the same so the CSN is 2 because the last change happened in release 2. This approach captures the essential information of the version number that is when the changes happen and which is the implementation of the *program*.

RSN	***1***	***2***	***3***	***4***	***5***	***6***
Versions of A	1.0	2.0	2.0	2.0	3.0	3.0
CSN of A	**1**	**2**	**2**	**2**	**5**	**5**
Versions of B		1.0	1.5	1.6	1.8	
CSN of B	**0**	**2**	**3**	**4**	**5**	**0**

Table 7-4 Example of change sequence number.

The CSN can assume also the value 0. This value doesn't map any system release. It's used to indicate that the *program* element is not present in that system release where it appears. For example, in Table 7-4 the *program* element B has this behavior. At release 1 it is not present, so its CSN is 0. At release 6 it has been removed from the system, so its CSN is 0. The choice has been to adopt the same system structure for all the releases. Such common structure is the most generic one, so that the structures of each release can be fitted to the common one. In this way the same structure is used for all the releases and when a system release doesn't have some modules, their CSN is filled with 0.

The advantage of the Change Release Database is that all the elements are numbered with the same notation, that is the RSN. In this way it is possible to make comparisons because all the entities are measured in the same scale. Moreover the use of null value for CSN allows to store also structural information about *programs* because it stores when the *programs* have been added or removed.

Specific programs have been written to convert the numeration of the case study and to set up the Change Release Database. The Table 7-5 provides an example of the database. It contains all the data of a *subsystem* (called A) which contains three *modules* (b, c and d).

Subsys	Module	Prog	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	b	1	0	0	0	0	5	5	7	7	7	7	11	11	11	14	14	14	14	14	14	14
A	b	2	0	0	0	0	5	5	7	7	7	7	11	11	11	14	14	14	14	14	14	14
A	b	3	1	2	3	4	5	5	7	7	9	10	11	12	12	14	15	16	17	17	19	20
A	b	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	17	17	17	17
A	b	5	0	0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	19	19
A	b	6	0	0	0	4	5	5	7	7	9	10	11	12	12	14	15	16	17	18	19	20
A	c	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	c	2	1	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	c	3	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	c	4	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	c	5	1	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	c	6	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	d	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	d	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	d	3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	d	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	d	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	d	6	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 7-5 An example of Change Sequence Database.

The numeration of all the elements is based on the RSN. In this way the mapping to a color scale is simplified because all the version numbers refer to the same scale and only the RSN values have to be mapped to the color scale. For the case study the values of RSN ranges from 1 to 20 (20 system releases). The CSN uses also the null value. So 21 values have to be mapped in colors. Black color is used for the null value. The other colors are shown in Table 7-6.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Table 7-6 Color scale for RSN and CSN.

7.3.2 Definitions

This section defines several terms that are used in the rest of the chapter. The definitions are specific for the case study as it has been described in the previous section. The terms are: "size", "maximum size", "normalized size", "change", "changing rate", " growing rate".

Sizes

The studies of M. Lehman and W. Turski [Turski96] [Lehman85] point out that for large software system the system size is measured by the number of modules it contains. So modules can be considered the smallest unit for measuring the size. In the case study the smallest unit of decomposition is the *program* element. Therefore, according to this definition and to A. Gurschler [Gursch96]

The size of each *subsystem* and *module* at a specific is defined as the number of *programs* it contains at that release.

For example, considering Table 7-5 the size of the *subsystem* A at release 1 is 14 *programs*. The size of *module* b at release 4 is 4 *programs*.

Each *subsystem* or *module* changes its size with time because *programs* are added or removed. For each *subsystem* or *module* we can define a maximum size:

The maximum size of each *subsystem* and *module* is defined as the maximum of all the sizes that the *subsystem* or *module* assume in all the releases.

For example, considering Table 7-5 the maximum size of *subsystem* A is 18 *programs* and the maximum size of *modules* b, c and d is 6 *programs*.

A particular choice has been made when describing the Change Release Database in Section 7.3.1. The same system structure is adopted for all the releases and the version number 0 is used to identify modules that have been removed or are not present in the structure of a specific release. In this way all the modifications are reported on the same common structure. The common structure is the most generic one where each *subsystem* and *module* assumes its maximum size. The size of a *subsystem* or *module* can be normalized using the maximum values. The definition is:

The normalized size of each *subsystem* and *module* at a specific release is the ratio between its size at that release and its maximum size.

For example, considering Table 7-5 the normalized size of *subsystem* A at release 1 is 14/18 = 77.8%. The normalized size of *module* d at release 1 is 6/6=100%, at release 2 is 0/6=0%.

Growing Rate

According to the definition of size given before, we can define:

The growing rate at release *n* is the number of *programs* of a particular *module* or *subsystem* which have been added between release *n-1* and release *n* minus the number of *programs* which have been removed between release *n-1* and release *n*. The growing rate is measured in *programs*/release.

For example, considering Table 7-5 the growing rate of *module* b at release 5 is 2 *programs*/release. The growing rate of *module* d at release 2 is -6 *programs*/release.

Changing Rate

According to the definition of version number given in Section 7.3.1, we can define changing rates for *programs*, *modules* and *subsystems*:

> For *program*: The changing rate between release n and release m is the number of times the *program* changes its version number between release n and release m.
>
> For *module* and *subsystem*: The changing rate at release n is the number of *programs* of a particular *module* or *subsystem* which changed between release *n-1* and release *n*. The changing rate is measured in *programs*/release.

For example, considering Table 7-5 the changing rate of *module* b at release 5 is 2 *programs*/release. The changing rate of *module* b at release 8 is 0 *programs*/release.

7.3.3 Navigation

A visual representation that supports the navigation allows the user to move the point of view for looking at the data. The 3-D representations developed in Section 7.2.2 offer such a capability. The three dimensional layout of objects increases the content of information and allows to navigate virtually within the system structure. Figure 7-14 shows all the structure of one release of the case study. All the *subsystems*, *modules* and *programs* are displayed.

Navigating the user can focus on interesting details, choosing the best view and he/she can have a virtual perception of the visualized structures. The display that generates the visualization has the following capabilities:

- Navigation support for 3-D space.
- A command allows to click on a graphical objects with the mouse to automatically move the viewpoint close to it.
- Clicking with mouse on a element the user can retrieve its attributes. Regarding the case study, clicking on a cube the user obtains the name and the version number of the module.

In Figure 7-14 all the 8 *subsystems* of the case study are displayed. The user can easily obtain the properties of modules (for example, the name) just clicking with the mouse on the objects without any other action. The navigation can allow to focus on particular *subsystems*. Figure 7-15 shows the details of several *subsystems*. Figure 7-16 is a zoom on the *modules* and *programs* of a *subsystem*. Figure 7-17 is a zoom on two *subsystems*.

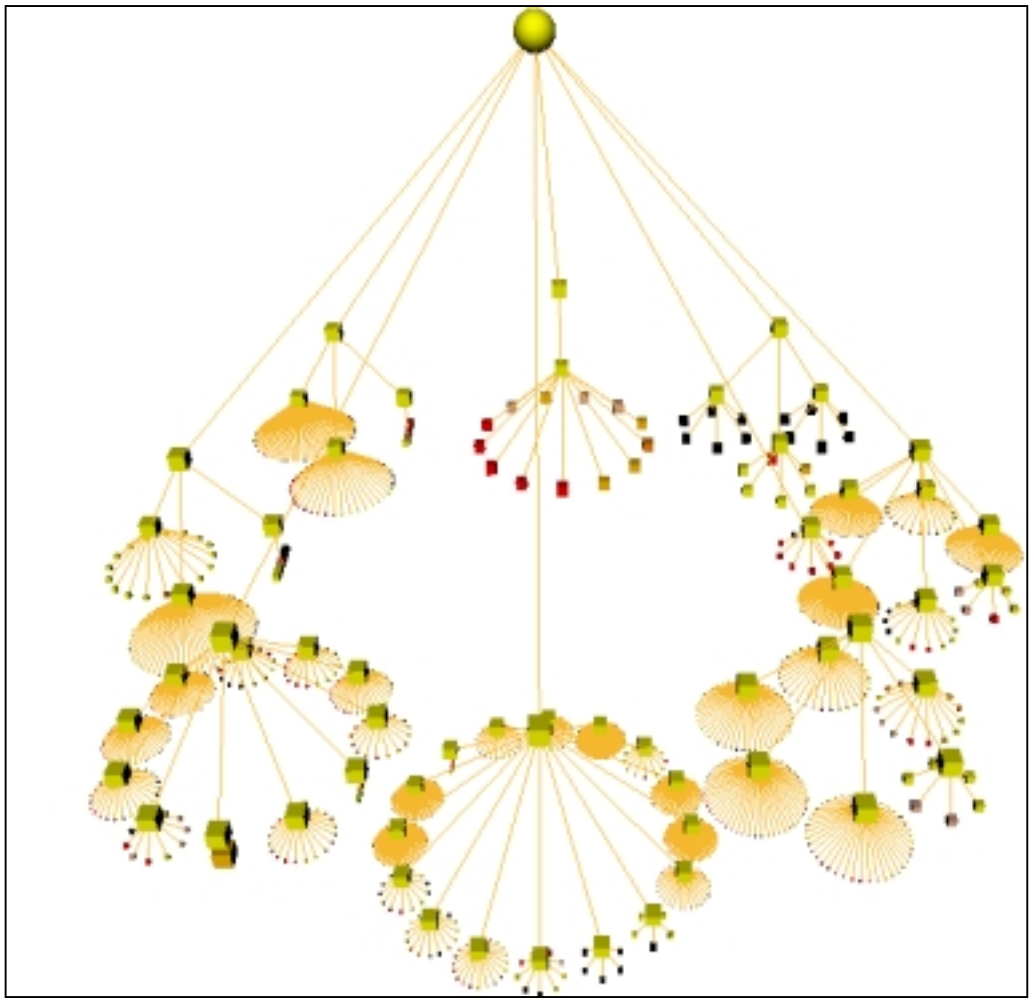

Figure 7-14 3-D visualization of the structure of the case study.

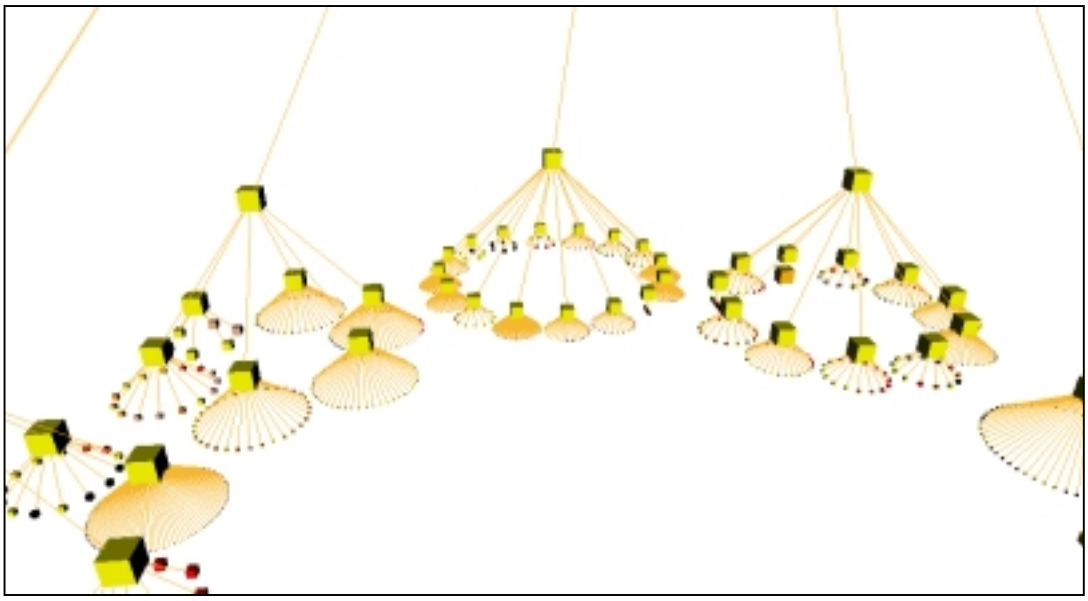

Figure 7-15 Navigation: focusing on *subsystems.*

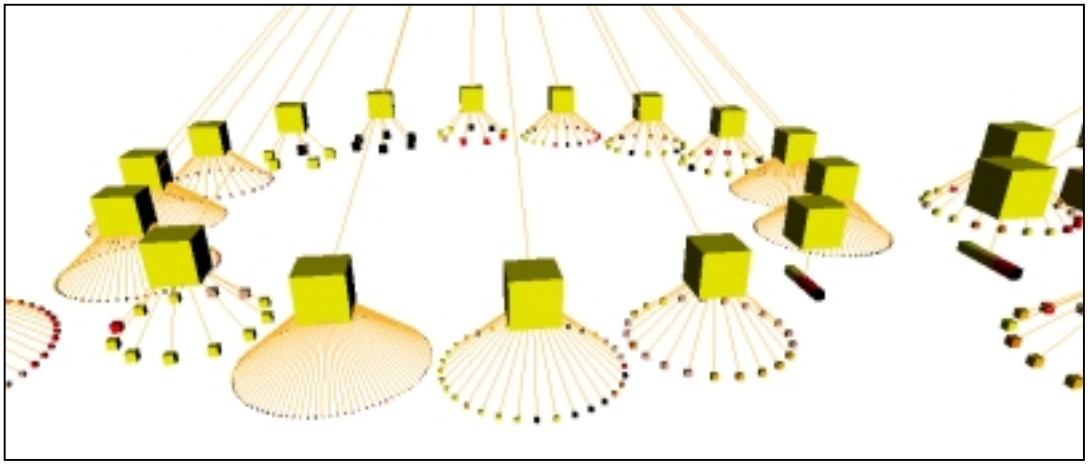

Figure 7-16 Navigation: focusing on *modules* and *programs*.

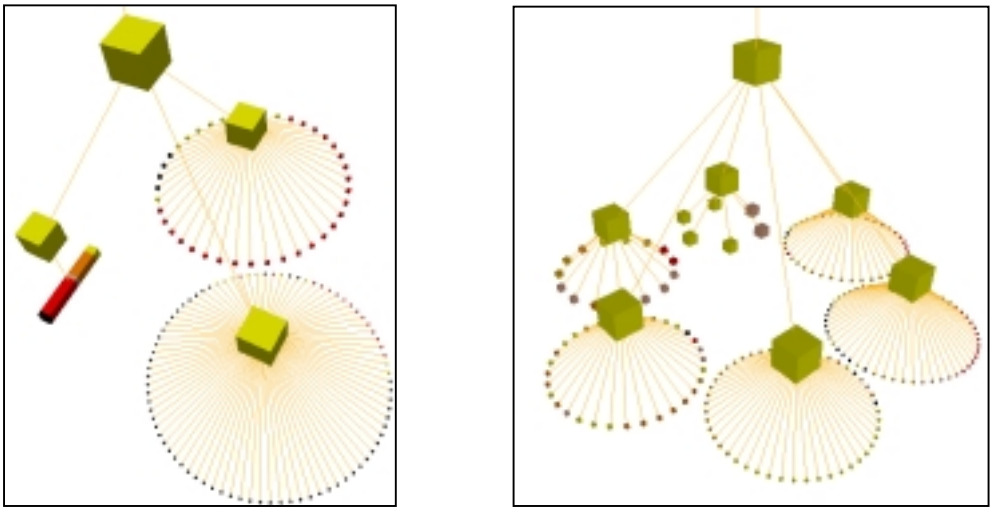

Figure 7-17 Navigation: focus on two *subsystems*.

Three dimensional display has the innate advantage of the virtual navigation of displayed information. Other advantages are mentioned here:

- Visual retrieve of data: to extract simple data (like name or version number) stored in the database the user has to set up several queries and to submit them to the database. This could be a boring and long process. The visualization and the capability of retrieving data from the graph helps the user to extract the information more quickly. This is possible using an interactive display described in Chapter 8.

- Global view: all the structure of one system release can be visualized in one view at any level of abstraction. In this way the user has all the data at his/her disposal. Moving or rotating the graphs it is possible to change the point of view and to focus on different sets of data.

- Local view: zooming allows the user to concentrate on specific data sets. For example, if the user finds something interesting or anomalous he/she can zoom on the problem and then go back at the point where he/she was before zooming without losing the context.

- Visual aid: when understanding abstract information our minds create visual representation to simplify the process. Visualization can provide it for the viewer and relieves his/her mind. In this way imagination and creativity are free of addressing new ideas.

- Easy of use: 3-D displays and navigation are easily understood because they take advantage of the innate perception that humans have of space. The training time for new user can be remarkably less [Chu98].

7.3.4 Visualizing large volume of data

The problem of visualizing large volume of data is directly addressed in this thesis. 3-D and 2-D graphs, even if displayed in a 3-D space, have the limitations that they become incomprehensible when visualizing large sets of data. Specific solutions have to be adopted. The percentage bar described in Section 7.2.2 is the graphical aid that offers a reduced representation.

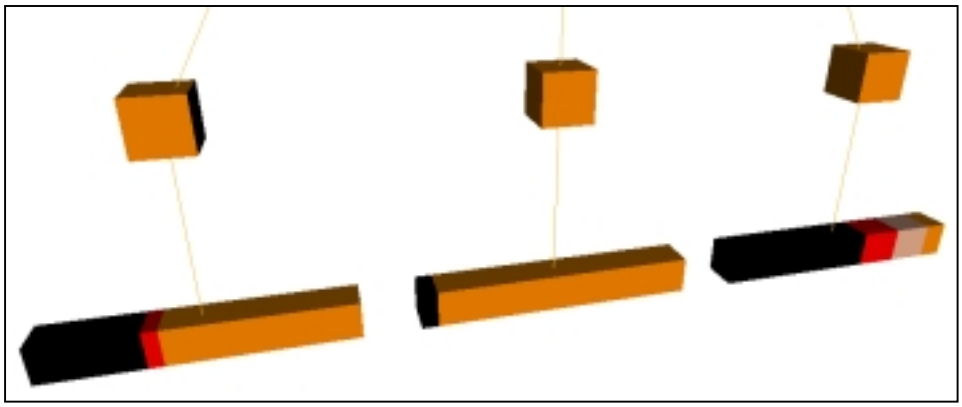

Figure 7-18 Example of percentage bars.

Figure 7-18 shows an example of percentage bars. The picture visualizes the structure of a *subsystem* at system release 3. At that release it contains three *modules* and 254 *programs*. It is one of the biggest *subsystems* in all the system, at release 20 it contains 504 *programs*, that is the 21% of all the *programs*. The program level is represented using percentage bars. Each bar gives the immediate information about the distribution of the attribute within the *modules*, that is how the versions numbers are distributed over the *programs*. The user can obtain the percentages interactively clicking with a mouse. The percentages are reported in Table 7-7. The first row represents the leftmost bar, the second the bar in the middle and the third the rightmost bar.

	Color	Percentage	Number of *programs*
First Bar (First *module*)	Black (not present)	33 %	20
	Red (version 3)	5%	3
	Orange (version 3)	62	38
Second Bar (Second *module*)	Black (not present)	6 %	1
	Orange (version 3)	94 %	16
Third Bar (Third *module*)	Black (not present)	61 %	263
	Red (version 1)	16 %	67
	Pink (version 2)	15 %	64
	Orange (version 3)	8%	36

Table 7-7 Percentages of Figure 7-18.

7.3.5 Visualizing history

The use of historical information is the innovative and original contribution of the Software Release History Analysis. The approach adopted in this thesis supports its visualization as described in Section 7.2.4. Two types of graphs are available. This section focuses on the Multiple 2-D tree which provides the best characteristics of visualization for historical analysis. Figure 7-12 and Figure 7-13 are two examples. They show the same system, the former without percentage bars the latter in the reduced representation.

The Multiple 2-D tree offers a good representation of historical data. All system releases are presented successively and the user can navigate through the trees to evaluate the changes between one release and the next. The graph can improve its layout to present the data in a better way. The original idea is to compact the graph so that each tree is close to the next one. Then, looking at the graph from the bottom, a 2-D view is obtained that shows the historical information in a compact way. This process of arranging the graphs is presented in Figure 7-19, where the graph of a *subsystem* is displayed.

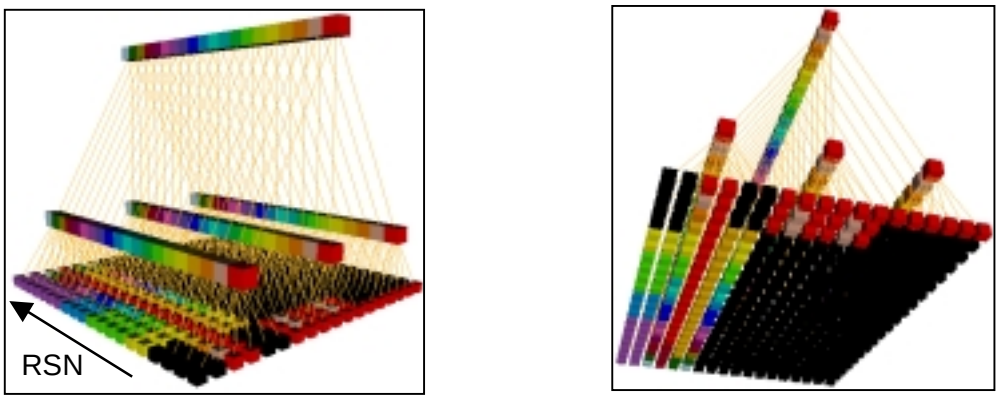

Figure 7-19 Rotating the Multiple 2-D Tree.

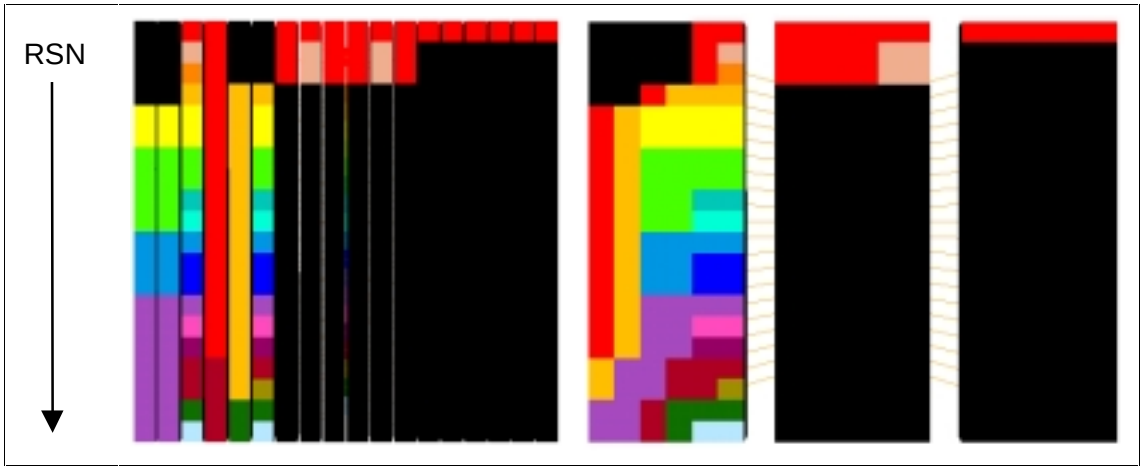

Figure 7-20 2-D with modules (left) and with percentage bars (right).

The effect of compacting and rotating the graph is shown in Figure 7-20. The picture on the left shows the *programs* of the *subsystem*. The picture on the right shows the same data using the percentage bars. Those pictures are a visualization of the data reported in Table 7-5. This 2-D representation is a powerful view for examining historical information and it takes the full advantage of the use of color. The features of this view are mentioned below:

- All the history is visualized in a compacted view so that the user can compare different releases.
- The main changes in the evolution are easily to detect because they are represented by big changes in color.
- The distribution of an attribute is visually perceived by region filling and colors.
- It is possible to focus on a single release and then to move the examination immediately on the next one without changing context.

Figure 7-26 shows the 2-D representation for all the *subsystems* of the case study. The time line is the same of Figure 7-20. The *subsystems* are

named using capital letters from A until H. Some observations that can be extracted by this view are reported in Section 7.3.8.

7.3.6 Examining the history

The visual representations presented in the previous Section 7.3.5 are a powerful instrument for examining the Software Release History. They show how an attribute is distributed over the modules. From this distribution several observation can be visually identified. These observations are described in this section. All the discussion is focused on the data and on the color scale introduced in Section 7.3.1. Only 2-D representations explained in Section 7.3.5 are considered here.

Changes of version

Changes of attribute, like version number, can be identified by changes in colors. Figure 7-20 is the visualization of Table 7-5. The data belong to a *subsystem*. We can focus on two *programs* which have a particular behavior. They are *program* 3 and 4 of *module* b. Their values and representation are in Table 7-8. Their behavior is different: *Program* 3 has an high changing rate. Instead, *program* 4 has a low changing rate. This behavior is immediately perceived by the changes of colors. The colors of *program* 3 change very often, the colors of *program* 4 are constant, except one variation. In this way colors are a visual instrument for identifying *programs* with a different behavior.

RSN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
b.3	1	2	3	4	5	5	7	7	9	10	11	12	12	14	15	16	17	17	19	20
b.4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	17	17	17	17

Table 7-8 Example of high and low changing rate.

Normalized size

The choice of adopting a common structure for all the releases (as described in Section 7.3.1) allows to visualize the normalized size of modules. According to the definition of CSN, a null value means that a *program* is not present in the structure and a not-null value means that the *program* occurs. Black color is associated to the null value of CSN. So it is possible to divide the color scale in two sub-regions characterized by a black color and by a not-black color. The former is associated to the absence of the *program*, the latter is associated to its presence. The normalized size is represented by the percentage of *programs* whose color is not black. For example, Table 7-9 shows the *programs* of *module* b from release 1 to 5 as given in Table 7-5.

Each row is a system release and each column is a *program.* At release 1, 2 and 3 the size of *module* b is 2 *programs*, then at release 4 it changes to 4 *programs* and finally it increases to the maximum value 6 *programs* at release 5. The normalized sizes are 33% at release 1, 2 and 3, 67% at release 4 and 100% at release 5. These data can be visually perceived in the picture. For each release the amount of not-black color regions is proportional to the normalized size.

RSN	1	2	3	4	5	6	Normalized Size
1							33%
2							33%
3							33%
4							67%
5							100%

Table 7-9 Visualizing the normalized size.

The same visual information can be extracted when displaying with percentage bars. Figure 7-21 shows the same *programs* as Table 7-9.

Figure 7-21 As Table 7-9 with percentage bars.

Growing rate

The changes of the normalized size of a *module* with time can give an indication of the growing rate of the *module*. Figure 7-22 shows the *programs* of three *module* with the percentage bars. Picture (a) shows a *module* with an high growing rate, picture (b) shows a *module* with a null growing rate and picture (c) a *module* whose *programs* disappear form the system. The size of the black region represents not-present *programs*. The modification in size of this region is an indication of the growing size. If the black region diminishes with time, the *module* is adding *programs*. If the black region is not present, it means that the *module* has reached the maximum size. If the black region increases with time, the *module* is removing *programs*.

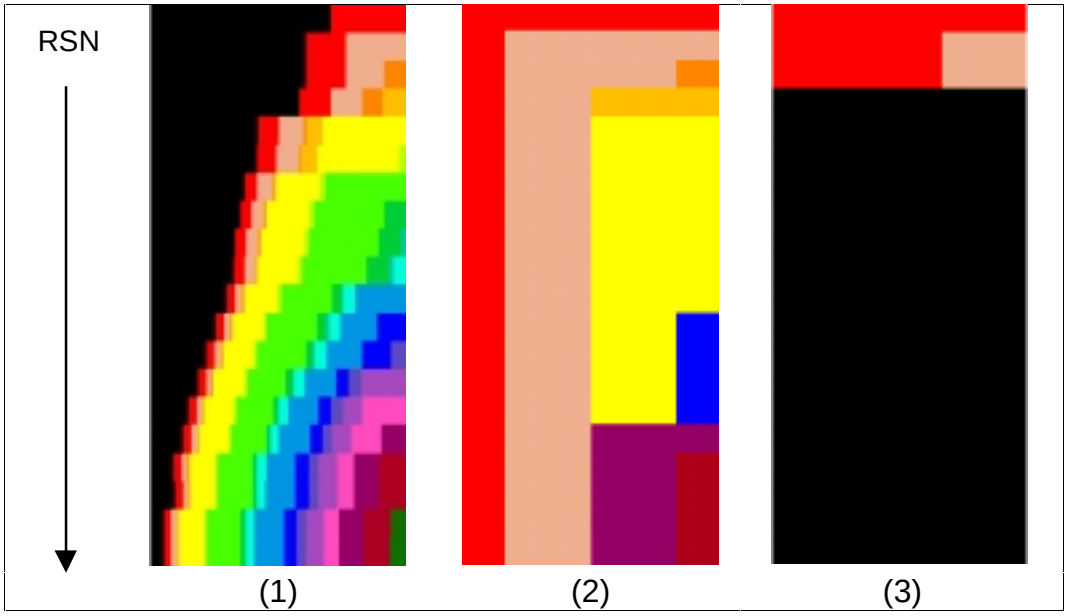

Figure 7-22 Visualizing growing rate: high (1), constant (2), negative (3).

Changing rate

Changes of version number are visualized as changes in colors. This property allows to have a qualitative and intuitive indication of the changing rate. High changing rate is identified by regions where colors change very quickly. Low changing rate is identified by regions with plain color. Figure 7-23 shows three different situations. In picture (a) the *module* has a very high changing rate, almost all the *programs* change their version number. In picture (b) changing rate is very low, only at release 19 and 20 some *programs* change their version. In picture (c) the *module* has an intermediate behavior, some *programs* are often changing and others never change.

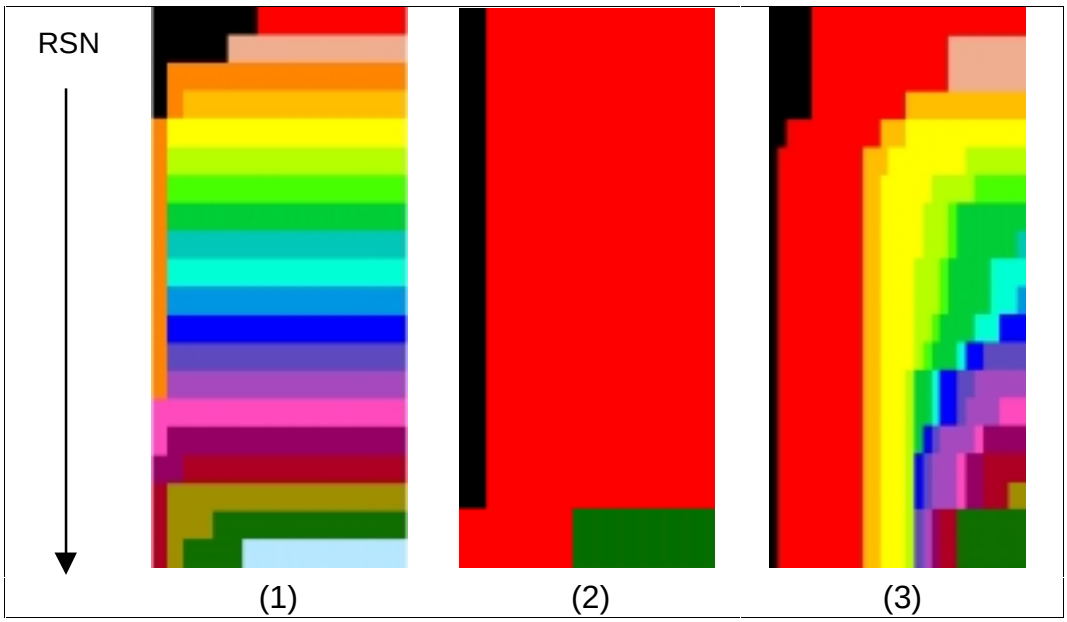

Figure 7-23 Visualizing changing rate: high (1), low (2), both (3).

Observations on the evolution

2-D representations introduced in Section 7.3.5 are a powerful instrument for examining historical evolution. For each *module* 2-D representations visualize the version numbers of its own *programs*. Such visualizations show how the *module*'s components have changed with time. Therefore they report the historical evolution of each *module* in terms of changes of its own *programs*. The examination of how the changes are distributed over the releases, can lead to identify the main modifications to which the *module* has undergone during its evolution. This section provides an example. Figure 7-24 and Figure 7-25 visualize the *module* under examination: the former uses percentage bars the latter shows all the *programs*. The time line is measured in RSN and its values are shown. The example refers to the third *module* of *subsystem* G in Figure 7-26.

The *module* contains 48 *programs*. In Figure 7-24 each row represents a system release and it is visualized with a percentage bar. Therefore each row visualizes the percentage of *programs* which have the same value of attribute, i.e. the same version number. In Figure 7-25 each row represents a system release and each column visualizes the version numbers of a *program* element that belongs to the *module*.

At release 1 (first row) all *programs* have the same version number that is the version number 1 (red color). In Figure 7-24 at release 2 (second row) 96% of *programs* have version 2 (pink color) and the rest (4%) has version number 1 (red color). This means that the major part of *programs* (96%) have changed their implementation and therefore the *module* has been largely modified. There are several reasons that can motivate such behavior. Motivations can be found by direct inspection of the code or of the *module*'s documentation. For example, the *module* could have been restructured, or, to add a new functionality, programmers had to modify many parts of it. To identify the small percentage of *programs* which doesn't change, Figure 7-25 has to be used. It shows that on the second row (release 2) the first *program* from left and the sixth *program* from right have a red color, i.e. version number 1. In this way the representation of Figure 7-25 allows to identify the outliners easily.

Considering Figure 7-24 at release 4 (fourth row), another major modification is present which influences the 85% of *programs*. This is visualized by the large orange zone. At the same release some *programs* (4%) are removed from the *module* (black zone). The inspection of Figure 7-25 allows to identify which *programs* have been removed (the second and the sixth from left).

At release 8 the *module* has its last major modification. In fact, in Figure 7-24 at release 8 (eighth row) the large dark green zone shows that many *programs* have changed their implementation. Then from release 8 until release 20 many of these *programs* maintain their version number: for each release from release 8 to 20 the green color zones are the biggest ones.

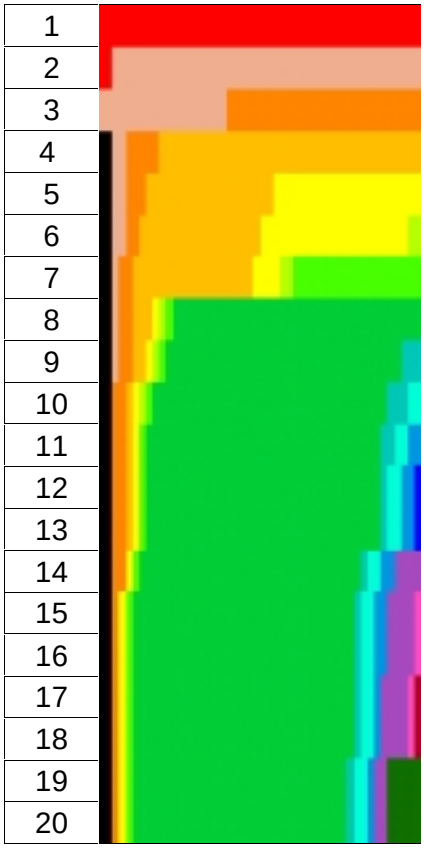

Figure 7-24 Visualizing history with percentage bars (time line in RSN).

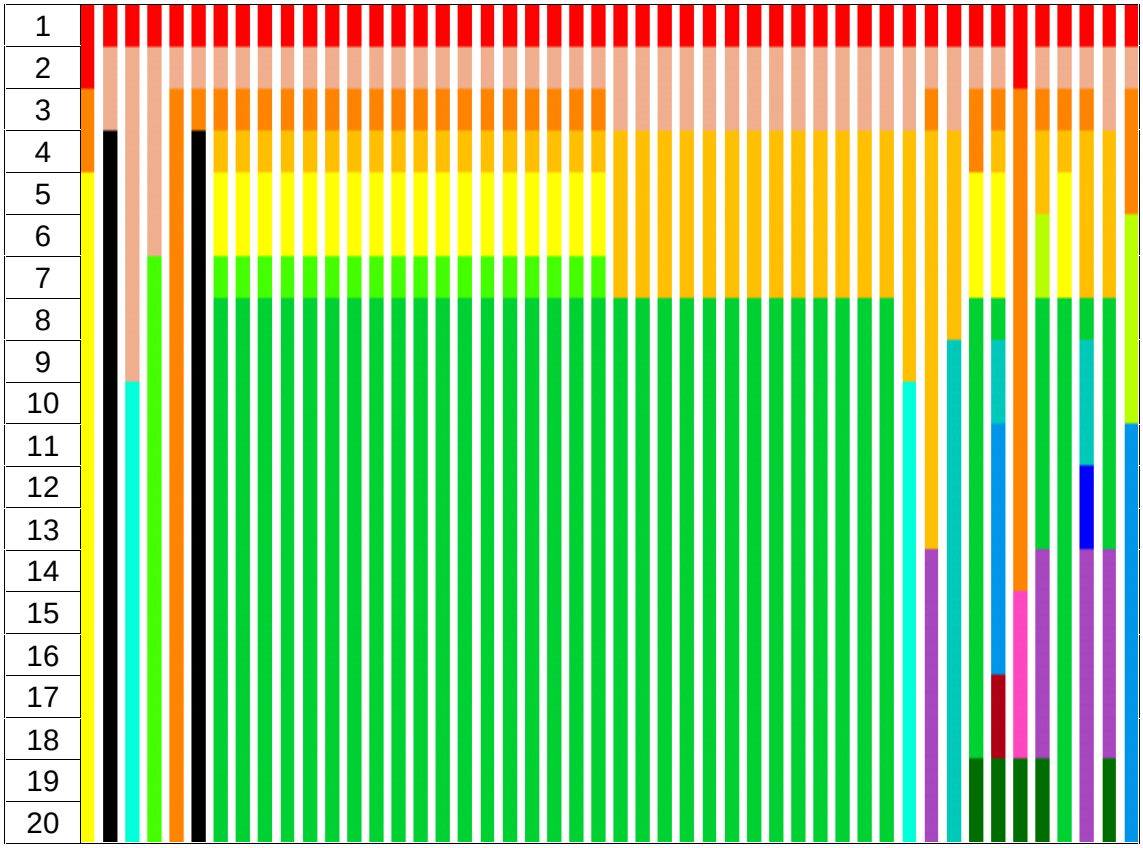

Figure 7-25 Visualizing history with modules (time line in RSN).

Between release 8 and 20 a small amount of *programs* changes its version number: in Figure 7-24 this is reported by the regions on the right colored with blue, purple and dark green. The *programs* change at releases 9, 10, 11, 12, 14, 15, 17, 19. Looking at Figure 7-25 it is possible to identify these *programs*. In particular, there are five *programs* whose behavior is anomalous. They are the second, the third, the fifth, the seventh and the eighth *program* from the right. All these *programs* have a common behavior: they change their implementation at release 8, maintain it for several releases and then they change it other times. The modification made at release 8 is successful for many *programs*. For the detected *programs* the modification is not so successful because after several releases they need other changes. Several reasons could be produced: the modification at release 8 has not been correctly implemented or the detected *programs* have particular problems and need to be restructured. Only a direct inspection of the source code or of the documentation can verify or not these hypotheses.

On the basis of the previous examination several observations can be made about the evolution of the *module*:

- Two *programs* have been removed at release 3 and they have been identified.
- Three major modifications that influenced a large part of the *module* have been discovered at release 2, 4 and 8.
- The major modification made at release 8 has not been followed by other major changes. The modification is accepted by many *programs*.
- From release 8 to release 20 the *module* is mainly constituted of *programs* which have been implemented at release 8.
- Several *programs* have been identified because the modification at release 8, that is accepted by many *programs,* involves successive changes.

Patterns

As mentioned in Section 6.5 one advantage of visualization is, that it enables the humans' pattern matching skills. This ability relies on the human visual system which is able to detect regions characterized by repetitive use of same shape, same color, same filling or same texture. Identification of patterns is needed to discover dependencies and relationships between system elements. An example can be identified in Figure 7-26 where all the *modules* of the case study are visualized using the 2-D representation described in Section 7.3.5.

Considering the first two rows of the picture, we can identify three *modules* which have the same color filling. They are the last *module* on the right of the first row, the fifth and the sixth *modules* on the second row. The common attribute is that their representations have a big region, filled with the same pink

color. This region begins at release 2 extends until release 20. This means that for each *module* many *programs* have the same version number 2 (pink color) and these *modules* don't change in all the releases. This observation could lead to identify relationships or commonalties among the *programs* or could lead to identify more generally that the *programs* have the same behavior.

7.3.7 Comparisons

As described in Section 7.1 a requirement for a visual representation of data is to support comparisons between different data sets. Three types of comparisons are required. In this section they are revisited to prove, that the adopted approach can support them.

Comparisons among modules

2-D graphs and 3-D graphs allow to visualize the structure of the system. With such representation the viewer can navigate through the data to compare all the modules of the system. Different views of the same configuration of modules can make such comparisons easier.

The use of color helps in the way it moves the process of comparing from a cognitive level to a perceptive level. Values of an attribute are represented with colors. Comparing the colors the viewer automatically makes comparisons of values.

Comparisons of different measures

The case study considered has the limitation that it contains only data about version numbers. Therefore it is not possible to investigate this capability. The idea is that comparing the same modules visualized with different attributes should be possible to identify commonalties or differences. This idea is explained in Section 9.3.

Comparisons of different releases

The representations obtained in Section 7.3.5 are the solution proposed for examining the historical information. All the historical evolution of a *module* can be visualized in a compact form that shows the changes of its *programs*. Such view allows to compare same *programs* in different releases and the historical evolution of different *modules*. This second feature is exemplified in next Section 7.3.8 when comparing different *modules* and *subsystems* of the case study.

7.3.8 Observations on the case study

The purpose of this section is to examine the visualization reported in Figure 7-26. It shows all the *subsystems* of the case study, they are labeled with capital letters from A to H. For each *subsystem* its *modules* are visualized using

the 2-D visualization with percentages bars as described in Section 7.3.5. For each *subsystem* the *modules* are numbered from left to right. The *subsystem* A contains two rows of *modules*, the first row contains the *modules* from 1 to 8, the second line the *modules* from 9 to 16.

Several observations can be produced examining Figure 7-26. The purpose is to show how the process of examining the Software Release History can take the advantages of the visualization. In particular, the purpose is to show how qualitative observations can be extracted. It is not a purpose to extract all the observations and to interpret them. The observations are divided by *subsystems*.

***Subsystem* A**

Module 16 is only present in the first system release. In fact, at release 1 (first row) all its *programs* have version number 1 (red color). Then for each release from 2 to 20, the percentage bar is black. This means that all the *programs* of the *module* are not present.

The major part of *modules* is characterized by a low growing rate. High growing rates are localized in the first releases, then *modules* reach a stable size. The sizes of the black regions are an indication for growing rate and normalized size as described in Section 7.3.6. *Modules* 3, 4, 5, 8 and 14 increase their size at the first releases, then the sizes become stable. They have an high growing rate at the first releases and then the growing rate reduces to a null value. *Module* 9 has a modest growing rate, then it becomes stable. *Module* 1, 6, 7 and 13 have a negative growing size, this means that they reduce their size with time. The size of *modules* 2 and 10 increases with time and then decreases in the last releases.

High changing rate can be identified by a zone with high color changes, instead of low changing rate that are identified by plain color zones. The *modules* 1, 2, 5, 7, 9, 11, 13 and 14 are characterized by big plain color regions, therefore their changing rate is quite low.

Release 3 is associated to the orange color. At releases 3 *modules* 2, 3, 5, 6, 9, 12 and 15 contain a large orange zone. This means that at release 3 many *programs* change their version number. For example, at release 3 *module* 5 adds many new *programs* and a big orange regions begin. This common behavior may be due to the fact that the *subsystem* has been largely modified and that this change interested many *modules*.

Release 5 is associated to yellow color. At release 5 *modules* 1, 2, 3, 4, 7, 9, 10, 11, 12 and 15 contain a large yellow zone. This means that at release 5 many *programs* have been modified, as happened at release 3. This may be due to a large modification of the whole *subsystem*.

Module 5 is characterized by the fact that the major part of its *programs* is added at release 3 (orange zone) and then many of them don’t change their version number anymore (the big orange zone extends until the last release).

This reveals that the programmers made good choices when they added the *programs* because these *programs* don't require any modifications for all the 20 releases.

Module 7 has almost the same behavior of *module* 5 just described. Three zones can be identified: red (release 1), yellow (release 5) and bright green (release 6). The red zone spans from release 1 until release 5 during whom the size of the *module* is unstable. At release 5 many *programs* change, this is identified by the small yellow zone. At releases 6 many *programs* (almost an half of all the *modules*) change their version number. And then this configuration is stable until the release 20, except for some changes identified by the small multi colored zone on the right of the picture.

Module 12 and 13 have almost the same behavior. Their representations have a big zone of pink color (release 2). This means that many *programs* have been changed at release 2 and then many of them maintain this version number until release 20.

***Subsystem* B**

Representations of *Modules* 7 and 8 contain a big red zone which extends until release 20. Red color is associated to version number 1. This means that a large amount of *programs* doesn't change from release 1 until release 20. This fact underlines that good choices have been done when the *programs* have been created, in fact these *programs* don't require any changes for all the 20 releases.

Module 9 contains a big orange zone. Orange is associated to release 3. A large amount of *programs* have been changed at release 3 and don't change anymore until release 20. It is the same behavior that *modules* 7 and 8 have.

The rest of *modules* is characterized by a moderate changing rate. This is revealed by the presence of zones where colors change frequently.

***Subsystem* C**

Module 1 has an almost stable size and it is mainly constituted of *programs* with version number 1 (big red zone).

Module 2 has an high growing rate at release 13 (dark purple color) when many *programs* are added. New *programs* are added with version number 13 and then many of them maintain it until release 20. Many of these added *programs* don't change their version number. In the last releases *module* 2 is mainly constituted of *programs* at release 13 (dark purple color) and at release 1 (red color).

***Subsystem* D**

Subsystem D contains *modules* with the highest changing rate. Almost all *programs* of *modules* 1 and 2 change their version number in each release.

This fact is shown by the horizontal colored lines which span for almost all the *module* size.

Module 3 has an high growing rate but the changing rate is modest.

Subsystem D has an anomalous behavior because two *modules* have high changing rates and one *module* has high growing rate. Its behavior has been also detected in the work of Gurschler using statistical analysis [Gursch96]. He identified this *subsystem* as a candidate for reengineering.

***Subsystem* F**

All *modules* except 3 and 7 have an high growing rate. Except for *module* 4 their size becomes stable with time. *Module* 4 maintains an high growing rate for all the releases.

Modules 3 and 7 have large plain color zones which mean that the changing rates are low. The growing rate is null, except for *module* 3 at the last releases.

In *module* 2 a large modification is present at release 15 (bright purple color), when many *programs* have been changed. After this release the changing rate is quite low. This reveals that the changes that tormented the first releases have been set up at release 15.

At release 15 *modules* 1, 4, 5, 6 have the same big change of *module* 2. This is identified by the line of bright purple color at release 15. The modifications that lead to a stable situation for *module* 2 aren't so effective in these *modules*. In fact, in *module* 2 the big bright purple zone persists until release 20, showing that the changes at release 15 are maintained. Instead *modules* 1, 4, 5 and 6 don't maintain this change, because their purple zone diminishes (*modules* 4, 5, 6) or at least disappears (*module* 1) . Only *module* 6 reveals a large bright purple zone, but successive changes have reduced it. The cause of this behavior could be that some *modules* of the *subsystem* have been changed for the same reason, for example restructuring or redesign. But the modifications have been effective just for *module* 2, because they have been maintained util release 20.

***Subsystem* G**

All *modules* are characterized by low growing rates. Except for *modules* 1 and 2, after the first releases sizes are stable. The *modules* which have a stable size (3, 4, 5, 6), have a big change at release 2 (pink color).

In *module* 3 a big change is present at release 8 (dark green color). After this change many *programs* maintain the version number.

Module 4 is affected by a big change at release 7 (green color) which is replaced by a big change at release 9 (bright blue color). This last change is not

maintained by many *programs,* because after release 9 many *programs* change their version number.

***Subsystem* H**

Two *modules* of the system have been removed in the first releases. After release 5 (yellow color) the size of *module* 1 becomes stable. Its changing rate is high.

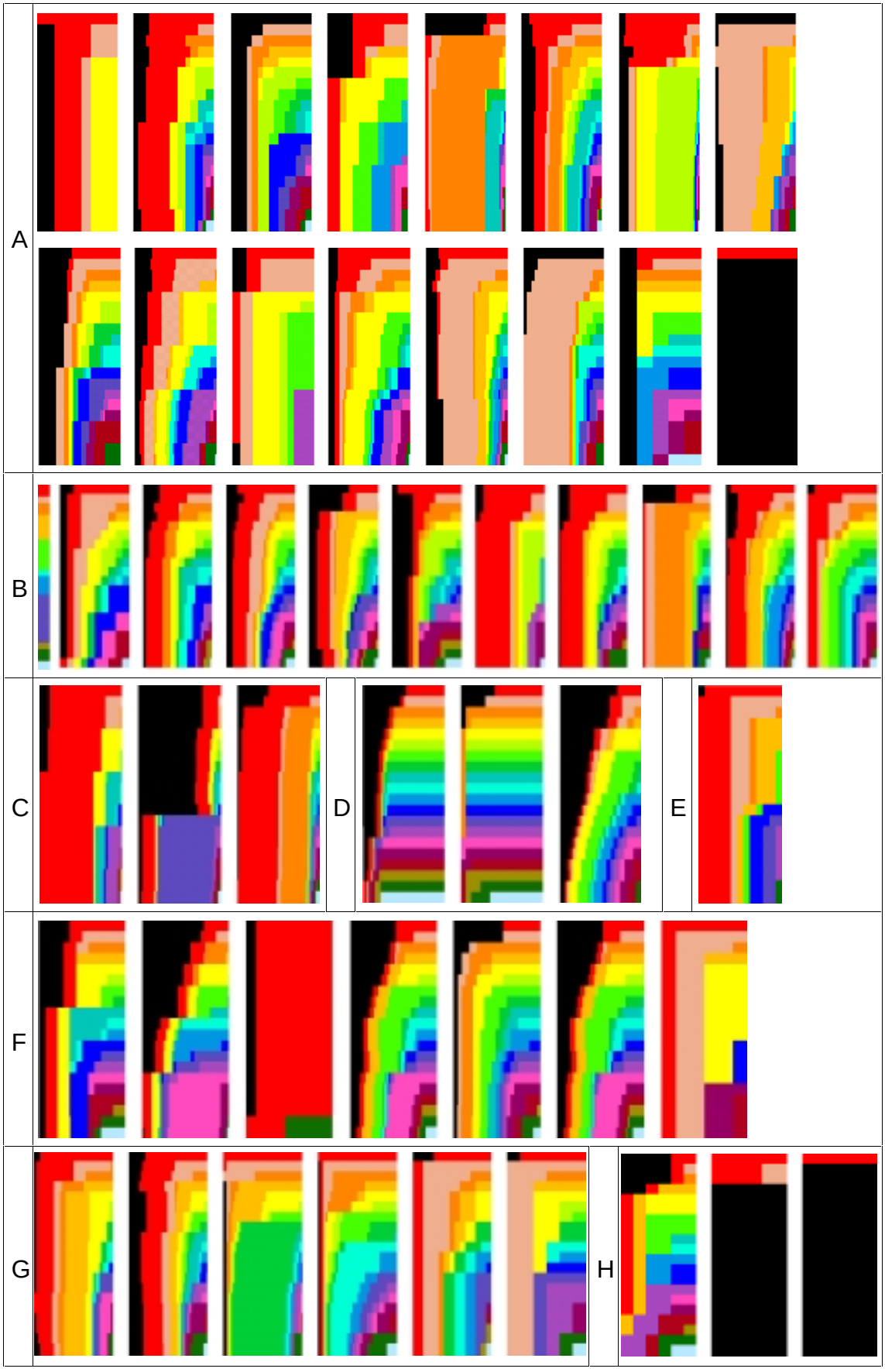

Figure 7-26 2-D visualization of the case study (RSN as in Figure 7-20).

7.4 Advantages of the approach

This section describes and summarizes the main advantages of the visual approach that has been adopted in this thesis.

7.4.1 Visualization

The main advantage of the approach is that it provides simultaneous visualization of three entities in one view. The entities are:

System structure

The abstract structure of a system at a generic release can be visualized with both 3-D and 2-D graphs. The graphical notation adopted (cubes and spheres for modules and lines for relationships) is intuitive and can be easily learned by software engineers and architects. The graphical representation adopted for the structures are close to the mental projections that humans' mind makes of abstract structures. For example, the hierarchical structures of the case study is visualized with trees that are the natural way we think of a hierarchical structure. The process of thinking of abstract information through mental images is alleviated, because the process is carried out automatically by a graphical system of visualization.

Measures

Each module of the structure is associated to a part of the software system. From a piece of source code several measures can be calculated such as version number, size, complexity. These values are the properties of the module. The approach used in this thesis allows to visualize with colors the values of modules.

Multiple releases

The third dimension is used as time coordinate to display the historical evolution of system structure and attributes of modules.

7.4.2 3-D visualization

The visual representation developed in this thesis uses three dimensional display. The main advantage of the third dimension is that it is possible to pack more information in one view. The advantages of the third dimension are:

- Three coordinates allow to visualize both system structure and historical evolution of the system. This is the main advantage that has been achieved for examining the Software Release History. The viewer can perceive both structural and historical information looking at one view. The viewer can also select the best prospective when focusing only on one information.

- Graphical objects have a three dimensional layout. The visual effect is a pleasure for humans' viewers. Rendering, shading and 3-D perspectives are the technologies that can simulate the reality to which humans' mind is usual. These graphical technologies can produce representations that are more natural to the humans' eyes.

- The viewer can navigate virtually in the graphical representation. This allows to choose the best view for the data he/she is interested instead of requiring the data to the database or changing the parameters of the graphs. The new representation can be easily obtained just rotating, zooming or moving the graph.

7.4.3 Coloring by measures

A module's attribute captures an essential information about its associated source code. Visualizing how the attribute is distributed over the modules, is useful for identifying anomalous behaviors or abnormal values of the attribute. The approach of this thesis uses colors for visualizing such distribution. The basic idea is coloring the system elements by their attribute's values. This is achieved mapping the range of values to a color scale. The advantages of this approach are summarized below:

- Attribute values are visualized together with system structure. In this way the values are in the same context of the elements that they belong to. When examining the structure, module's properties are immediately available and vice versa when examining the attribute distribution the structure is also displayed.

- Mapping numerical values to colors the process of comparing the data moves from being a cognitive task (i.e. numerical comparison) to being a perceptive task (i.e. visual comparison). This is the main advantage of using colors. Changes, commonalties, differences and patters can be visually detected.

7.4.4 Reduced representation

Visualizing the attribute values of a large set of modules the main problem is how to give an informative representation that would not be incomprehensible by the large amount of data visualized. The approach proposes the percentage bars. These graphical objects provide a visual representation of percentages instead of values. In this way it is possible to visualize a set of values of arbitrary size in a standard layout without being overwhelmed by the volume of data. For example, in Figure 7-26 all the modules of the case study are represented in a reduced and standard form independently of their own size. Percentage bars display the percentages using different sizes. For quantitative comparisons, size is the most effective perceptual data encoding variable [Eick97].

7.4.5 Visualization of History

The approach allows to visualize and compare multiple system releases. Two representations that are useful for studying the historical evolution are described in Section 7.3.5. They are obtained by an arrangement of the 2-D Multiple Tree graphs. Representations that use percentage bars report the major modifications that signed the history of a module and allow to discover anomalous behaviors. Representations that don't use percentage bars, visualize the historical evolution of each module's component and allow to precisely identify the outliners. An example is provided in Section 7.3.6.

7.4.6 3-D navigation

3-D Tree and Multiple 3-D graphs display the system structure using three dimensions. There are two main advantages: displaying large amount of data in one view and navigating through the data in 3-D space.

Instead of accessing directly the database to look for the data and to extract them, it is possible to navigate through its visualization and to have access to the desiderated data immediately. The visualization system proposed in Chapter 8 and developed in this thesis supports both functionality. The viewer can navigate the 3-D space for examining the structure and he/she can retrieve the data just pointing the mouse on a object and clicking on it. For example, Figure 7-14 visualizes the whole structure of one system release that is contained in the database. The lowest level consists of almost 2300 elements. The viewer has a global view of all the database in just one view. Then he/she can focus on a particular subsystem or can extract the values of the modules just clicking with the mouse.

Chapter 8

3DSOFTVIS VISUALIZATION SYSTEM

This chapter describes the tool that has been developed for supporting the graphical visualizations explained in Chapter 7. The purpose is to show the original ideas that have been applied for its development, and not a description of its functionality.

8.1 Overview

The tool developed to support the graphical visualization is called 3DSoftVis. 3DSoftVis is a World Wide Web-based application. World Wide Web (or simply the Web) is a technology that allows to export a set of resources to a community of users through a network, such as the Internet. The user can access the network resources through a highly portable display interface called Web browser. A Web browser allows to retrieve Web documents on the network and to show their content. A Web document can contain different types of resources. For example, Web documents can be written in HTML (Hyper Textual Markup Language). HTML allows to create documents that contain information in textual or graphical form. Recent technologies allow to write interactive Web documents. In our context the focus is on two new technologies: Java and VRML.

Java [Arnold96] is an object oriented programming language. A type of Java program is particular useful for writing interactive Web documents: *Java applet*. Java applets are programs that can run within Java-compatible browser. A Web documents can contain one or more Java applets. When a document containing a Java applet is loaded in a Web browser the program starts and executes its code. The applet can interact with the user and the Web document, can load other documents in the Web browser and can start other Java applet. Security limitations prevent the applet from accessing the machine where the Web browser is running, i.e. user's machine.

VRML (Virtual Reality Modeling Language) [VRML97] [Ames96] [Pesce95] is a platform-independent 3D modeling format for describing interactive 3D objects and worlds. VRML is capable of representing static and animated dynamic 3D and multimedia objects with hyperlinks to other media such as text,

sounds, movies, and images. VRML is designed to be used on a network context. Web documents containing VRML resources can be loaded in Web browsers if they have been enabled for displaying VRML data.

The integration of Java, VRML and Web browsers is a solution for developing visualization tools for 3-D graphical forms. Such tools are Web based system, platform independent and can be accessed through a network. In literature several studies are proposing such a Web based architecture for exporting information and data to a community of users [Eick98] [Brown97] [Santo97] [Rau97] [Lee97].

3DSoftVis has been implemented using the Web components just mentioned: Java, VRML and HTML.

8.2 Requirements

The problem discussed in this thesis is to develop a graphical approach for the examining the Software Release History. This problem is largely investigated in Chapter 6 and Chapter 7. This section extends the requirements listed in Section 7.1 with the ones for implementing the graphical system. They are stated and motivated below:

3-D graphical support

The system has to support the visualization of 3-D graphics and the 3-D navigation of them. The graphics to visualize are the ones described in Chapter 7.

User interface and multiple views

The tool needs a user interface for choosing the different types of visualization and for selecting the data to visualize. The user interface has to support multiple windows for displaying simultaneously different types of graphical views.

Access to database

The tool must access the database containing the Software Release History to query the data to visualize. The database has the same structure of the one represented in Table 7-5.

8.3 System Description

This section explains the system architecture and the user interface.

8.3.1 System Architecture

The system architecture, illustrated in Figure 8-1, is Client/Server. The server machine contains the Web server and the database of the release history. Web server exports several web documents in HTML format and the Java packages which contains the Java programs. Web documents contain Java applets that run the tool. The client is a web browser which is Java and VRML enabled.

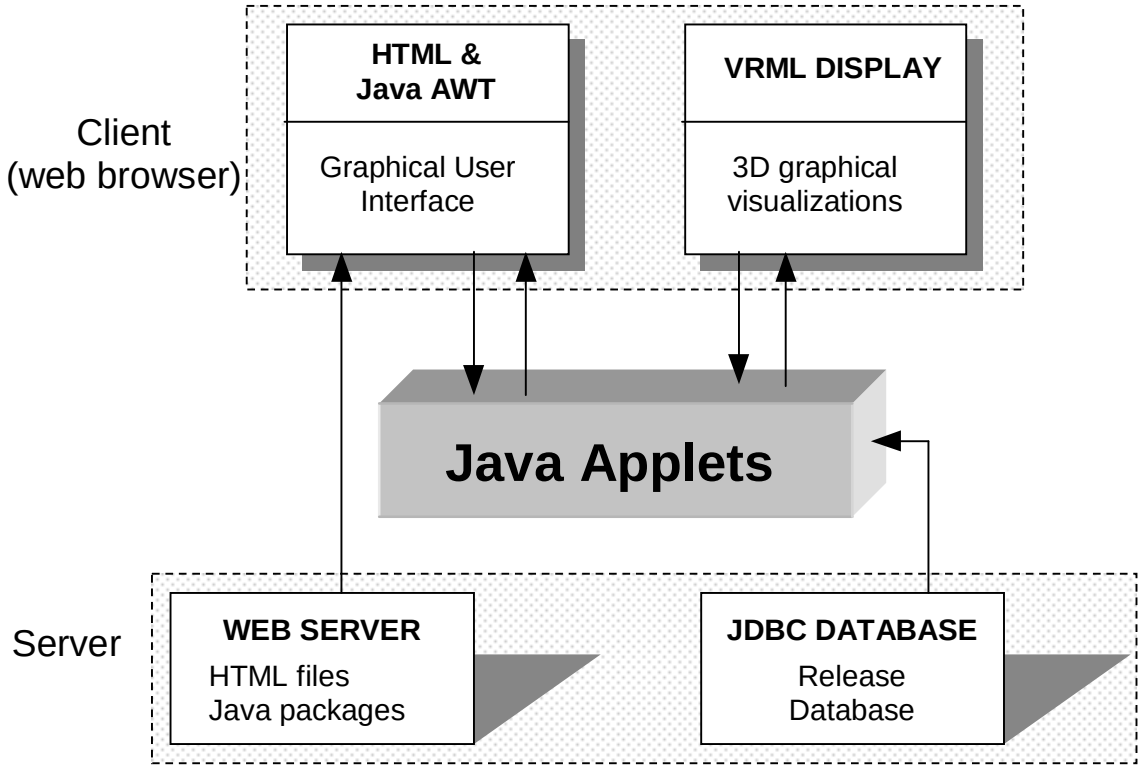

Figure 8-1 System Architecture of 3DSoftVis.

To start the tool the user has to access the web server through the network and has to load the main web document in the web browser. The web document, that is an HTML page, contains a Java applet. When the web document is loaded the applet starts. The applet shows the Graphical User Interface (GUI) that is used to interact with the tool. The applet opens also a connection to the server machine for accessing the database which contains the data to visualize.

All the interactions between the tool and the user are managed through the graphical user interface. It allows the user to select the data to visualize, to open multiple windows for visualizing different data, to manage the opened windows and to quit the program.

8.3.2 The Graphical User Interface

The user interface is implemented using the Java Abstract Window Toolkit (AWT) which provides common graphical components such as windows, menus, buttons. According to the requirements 8.2 the user interface must support multiple windows and must provide a way to select the data for displaying. It is composed of three elements: *workspace*, *view window* and *property window*. Figure 8-2 shows a snapshot of the tool. It displays the workspace window, two view windows and the property window.

Workspace

The workspace is the main frame of the tool. It comes up when the user loads the main web document as explained in Section 8.3.1. It allows to create, manage and close the windows that the tools use for the visualizations.

View window

The view window is a graphical object which is constituted of three elements: a VRML browser, a textual display and several buttons. Figure 8-3 shows a snapshot of it with the indication of the three elements. The view windows is used to show the 3-D diagrams that are described in Chapter 7.

The VRML browser is a component of the Web browser and allows to create and display three dimensional objects. The VRML browser allows the viewer to navigate in the 3-D space through a set of navigational commands. These commands are located in a toolbar in the bottom of the browser. Through the commands the user can change his/her viewpoint, zoom on details, rotate the displayed objects and select predefined points of view. The view windows displays the 3-D diagrams in the VRML browser. In this way the navigational support is automatically implemented in the VRML browser.

The textual display shows the data that the user retrieves form the 3-D diagram. The VRML browser allows the user to click with the mouse on the 3-D objects to extract their properties. For example, clicking on a 3-D cube the textual display shows the name and the version number of the associated elements. In Figure 8-3 the textual display shows the property of a *program* element of the case study.

The view window contains three buttons: Clear, Property and Close. The first button is used to clear the content of the textual display. The button Property opens the property window associated to this view window. The button Close is used to close the view window.

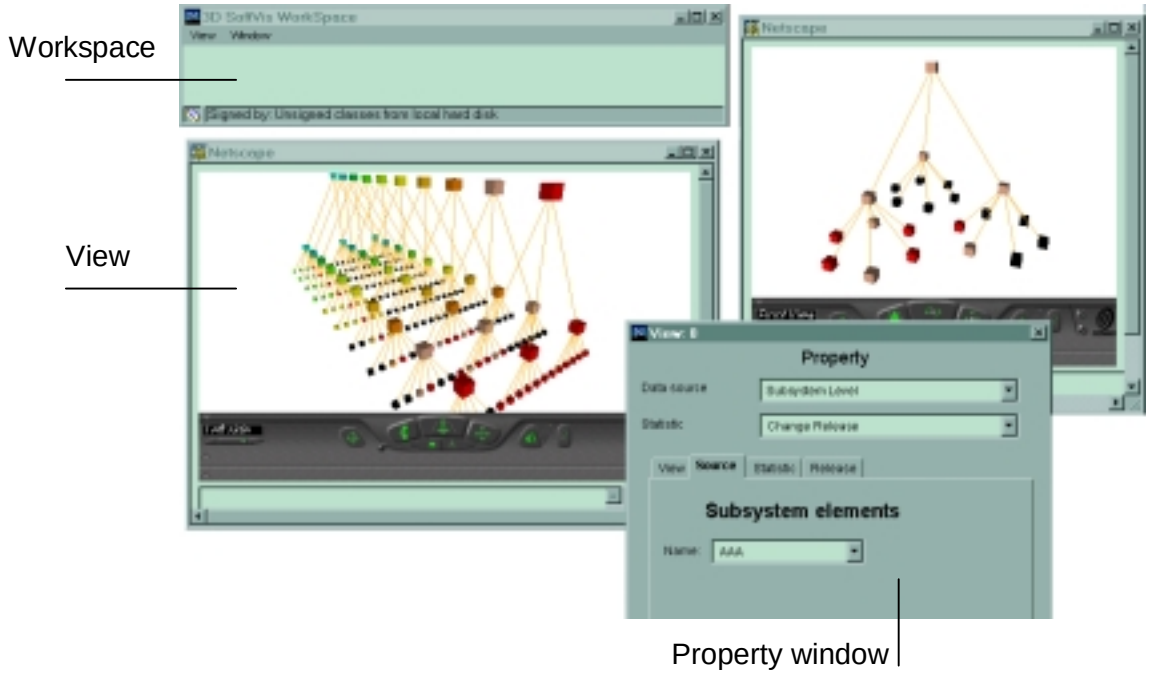

Figure 8-2 Snapshot of 3DSoftVis running within Netscape's Navigator™.

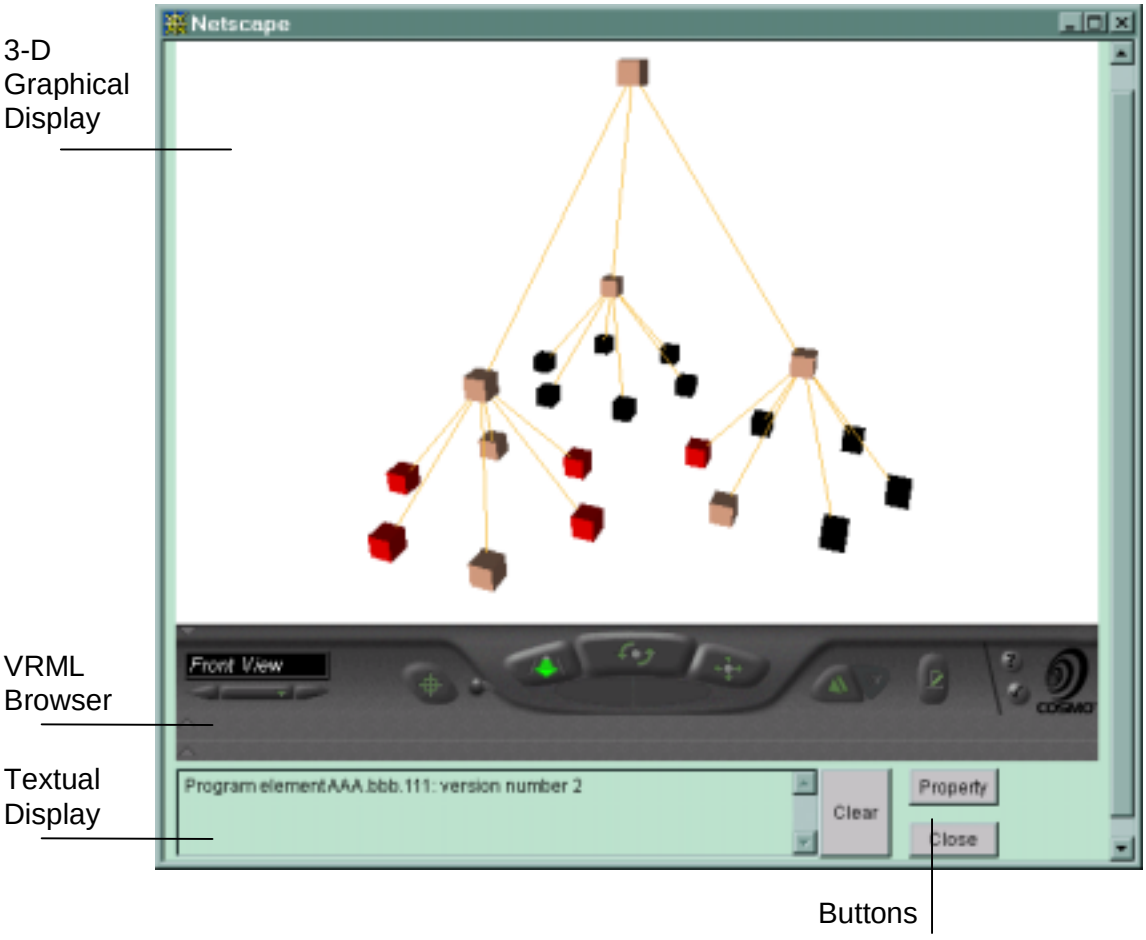

Figure 8-3 The View window.

Property window

To each view window a property window is associated. The task of the property window is to allow the user to configure the displayed graphics in the view window. There are four sets of properties that a user can set to configure a 3-D diagram:

- Layout and appearance of 3-D graphics: properties that regard how the 3-D graphics (2-D Tree, 3-D Tree, Multiple 2-D Tree and Multiple 3-D Tree) are displayed in the view. The properties are: size of elements, size of graphs, distance between multiple graphs and properties of percentage bars.

- Structure to visualize: these properties allow to choose which elements of the structure have to be visualized. The user can choose the abstract level and the names of the modules. For example, the user can choose to visualize all the *programs* of a particular *module*.

- Statistic to visualize: these properties are used to choose the statistic or the measure that has to be visualized. In particular its options allow to set up the mapping between the numerical values and the color scale.

- Releases to visualize: properties to choose which system releases the tool has to visualize in the 3-D diagram.

8.4 Advantages

The major advantages of 3DSoftVis in supporting the visual representation are discussed in Section 7.2.5. This section presents additional advantages that are emerged in this chapter.

8.4.1 Support of 3-D graphics

The VRML browser is the graphical engine of 3DSoftVis. It implements all the operations that are required to produced 3-D objects and to navigate the 3-D space: 3-D rendering, shading, coloring, rotating, zooming and moving. This choice has a variety of advantages including: cost (free), easy integration with Web-based components like Java, platform independence and high performance of graphical viewers.

8.4.2 Web-based application

The Web-based architecture makes the tool accessible from the network. A web browser, that is VRML and Java enabled, can run the tool just loading the main HTML page that is located on the server machine. This allows the data to reside in a central location (the server machine), but the analyses to be

conducted locally on the client machine. This flexibility provides several advantages:

- The user can start the tool without downloading the data and installing the application on his/her machine by himself.
- Readers are always working with the most up-to-date version of the data because data are centralized only in one location.
- Collaborative researches are encouraged because researchers can work on the same data and with the same application without requiring to be in the same location.
- A demo of 3DSoftVis can be published on the Web so that readers can confirm the results and conclusions reached in this thesis.

Chapter 9

CONCLUSIONS

9.1 Summary

The Software Release History tracks the evolution of a software system in terms of the changes that it undergoes during its life-cycle. Examining the historical evolution can lead to discover problems or shortcomings and to identify the modules that need restructuring or reengineering. The purpose of this Master's Thesis is to develop a visual representation for examining the Software Release History. The visual representation is an aid to the examination process. The issues of this problem were discussed in Chapter 6. The approach adopted in this thesis uses three dimensional diagrams. Each diagram can simultaneously shows three entities of the Software Release History: the system structure, an attribute that has been measured such as version number, size and detected bugs and the historical evolution. Colors are used to visualize the values of the attribute. Mapping the numerical values to a chromatic scale is the original key to achieve a perceptive solution of the problem. This approach is described in Section 7.2. The case study provides a large amount of data that can be used for testing the approach adopted. Section 7.3 showed how to use the visual representation for examining the data of the case study. The objective is to prove that the visual representation satisfy the purpose of this thesis. Finally in Chapter 8 the tool that has been developed for supporting the visual representation is described.

9.2 Summary of Contributions

This section summarizes the major contributions of knowledge that have been achieved in the present Master's Thesis. They are ordered for importance.

1. A visual representation of the Software Release History has been developed and its usefulness has been proved. The visual representation can show structural information, historical information and a measure that is calculated on the source code simultaneously.

2. The use of the third dimension has been investigated. Its major advantages are: packing more data in one view, 3-D navigation of graphics and graphical pleasure.

3. The use of color for representing the distribution of numerical values has been investigated. It reveals to be an effective perceptual aid for detecting patterns and outliners.

4. The visual approach adopted in this thesis proves that visualization can be a solution for understanding large and complex data sets.

5. A visualization tool has been realized for supporting the visual representations. Its development has investigated the use of Web technologies.

9.3 Future Research

In Chapter 6 and Chapter 7, when describing the methodology for examining the Software Release History, a generic approach has been adopted in prevision of further developments of the visual representation that are described here.

Examples and observations consider only one module's attribute, that is the version number of modules. This limitation is due to the database of the case study. The database stores only the version numbers of the system element. Future work should concentrate on the extension of the Software Release History Database of the case study, in order to include more detailed information about the modules. For each module it should include its size in LOC, detected bugs, complexity measures or other measures as described in [Fenton96]. Then, by setting up the mapping of the color scale, the visual representation can be used to visualize also these attributes.

The case study has a peculiar hierarchical structure. For this reason, the present work has adopted only tree structures (2-D and 3-D). Future works should be directed in investigating other structural representations that could be used for non-hierarchical architectures.

The intent of this thesis is to provide a visual representation for examining the Software Release History. Several observations about the case study have been produced. These observations are to be verified by using additional information such as bug reports, enhancement requests, design changes or by direct inspection of the source code.

A related work [Hajek98] addressed the problem of identifying module dependencies by detecting common patterns in the Software Release History. Future works should investigate the possibility of automating the task of detecting patterns. In particular two types of capability should be supported by the tool: visualization of known patterns and automated detection of patterns introduced by the users.

The developed tool, 3DsoftVis, is a web-based application. Such a application can be exported through the network and can be used by a set of users. This capability and its implications have to be carefully analyzed. For example, when examining software data as a part of a team, it could be useful to share the same data and to use the common application. The web-based application can also be the framework to exchange ideas and observations among the programmers. In this way collaborative researches can be encouraged.

Chapter 10

REFERENCES

[Ames96] A. Ames, D. Nadeau, J. Moreland, *The VRML Sourcebook,* John Wiley & Sons, New York, 1996.

[Arnold93] R. S. Arnold, *Software Reengineering*, Computer Society Press, 1993

[Arnold96] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996

[Baker94] M. J. Baker and S. G. Eick, Visualizing Software Systems, *AT&T Bell Laboratories*, 1994.

[Bennet91] K. H. Bennet, Automated support of software maintenance, *Information and Software Technology*, Vol.33, No. 1, Jan/Feb 1991, pp. 74-85, reprinted in Arnold R., S. *Software Reengineering*, Computer Society Press, 1993, pp. 59-70.

[Hajek98] K. Hajek, Detection of Logical Coupling Based on Product Release History, Master's Thesis, Technische Universität Wien, Vienna, Austria, May 1998

[Brown97] M. H. Brown and R. Raisamo, JCAT: Collaborative active textbooks using Java, Computer Networks and ISDN Systems, Vol. 29, No. 14, October 1997, pp. 1577-1586.

[Chikof90] E. Chikofsky and J. Cross, *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, Jan 1990, pp. 13-17

[Choi90] S. C. Choi and W. Scacchi, *Extracting and Restructuring the Design of Large Systems*, IEEE Software, Jan. 1990, pp. 66-

[Chu98] H. Chu and H. Koike, How does 3D Visualization Work in Software Engineering ?: Empirical Study of a 3D Version/Module Visualization System, *International Conference Software Engineering 98 (ICSE 98)*, 1998.

[Corbi89] Corbi T. A., Program Understanding: Challenge for the 1990s, *IBM Systems Journal*, Vol. 28, No. 2, 1989, pp. 294-306, reprinted in Arnold R., S. *Software Reengineering*, Computer Society Press, 1993, pp. 596-608.

[Eick96] Stephen G. Eick and Daniel E. Fyock, *Visualizing corporate data,* AT&T Technical Journal, January/February 1996, pp. 74-76.

[Eick96a] Thomas A. Ball and Stephen G. Eick. *Software visualization in the large,* IEEE Computer, April 1996, pp. 33-43.

[Eick97] Stephen G. Eick*. Engineering perceptually effective visualizations for abstract data*, In Gregory M. Nielson, Heinrich Mueller, and Hans Hagen, editors, Scientific Visualization Overviews, Methodologies and Techniques. IEEE Computer Science Press, February 1997, pp. 191-210.

[Eick98] S. G. Eick, A. Mockus, T. L. Graves and A. F. Karr, A Web Laboratory for Software Data Analysis, World Wide Web Journal, To appear (1998), http://www.bell-labs.com/user/eick/eick_bib.html

[Encyc94] John J. Marciniak, *Encyclopedia of Software Engineering*, John Wiley & Sons, 1994

[Fenton96] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Pratical Approach*, International Thomson Computer Press, Second Edition, 1996

[Fjeld79] R. K. Fjeldstad and W. T. Hamlen, Application program maintenance study: Report to our respondents, *Proceedings of GUIDE 48*, The Guide Corporation, Philadelphia, 1979

[Gall96] H. C. Gall and R. Klösch, *Improving Reusability of Legacy Applications through Object-Oriented Re-Architecturing*, Technical Report TUV-1841-96-07, Distributed Systems Group, Technical University of Vienna, Dec. 1996.

[Gall97] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. *International Conference on Software maintenance (ICSM '97)* (Bari, Italy), pages 160-6, M. J. Harrold and G. Visaggio, editors. IEEE Computer Society Press, September 1997.

(awarded the best paper of ICSM '97).

[GAO81] *Federal Agencies' Maintenance of Computer Programs: Expensive and Undermanaged*, AMD-81-25, U.S. General Accounting Office, Washington, DC, February 1981

[Garlan93] D. Garlan and M. Shaw, *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company, Volume I, 1993

[Ghezzi91] C. Ghezzi, M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 1991

[Gursch96] Alexander Oswald Gurschler, Software Architecture Assessment Based on Product Release History, Master's Thesis, Technische Universität Wien, Vienna, Austria, November 1996

[IEEE93] IEEE, IEEE Software Engineering Standards Collection, *Institute of Electrical and Electronics Engineers*, New York, NY, 1993.

[Jerd97] Dean Jerding and Spencer Rugaber, Using Visualization for Architectural Localization and Extraction, *IEEE Transactions on Software Engineering*, Aug. 1997, pp. 56-65.

[Keller96] T. Keller, Change Costing in a Maintenance Environment, *International Conference on Software Maintenance (ICSM '96)*, pp. 130-131, Monterey, California, November 1996

[Lee97] C. Lee, T.-y Lee, T.-c Lu and Y.-t. Chen, A World Wide Web based distributed animation environment, Computer Networks and ISDN Systems, Vol. 29, No. 14, October 1997, pp. 1635-1644.

[Lehman71] L. A. Belady and M. M. Lehman, Programming system dynamics or the metadynamics of systems in maintenance and growth, *Res. Rep.* RC3546, IBM, 1971, reprinted in M. M. Lehman and L. A. Belady, Program *evolution*, Academic Press, London and New York, 1985, pp. 99-122.

[Lehman76] Lehman M.M. and F. N. Parr, Program Evolution and Its Impact On Software Engineering, *Proceedings of the 2nd Conference on Software Engineering*, San Francisco, October 1976, pp. 350-357, reprinted in *Program evolution*, Academic Press, London and New York, 1985 pp. 201-220.

[Lehman78] L. A. Belady and M. M. Lehman and, Characteristics of Large Systems, *MIT Press* 1978, reprinted in M. M. Lehman and L. A. Belady, Program *evolution*, Academic Press, London and

New York, 1985, pp. 99-122.

[Lehman82] M. M. Lehman, Program Evolution, *Symposium on Empirical Foundation of Computer and Information Sciences*, 1982, Japan Information Center of Science and Technology, reprinted in M. M. Lehman and L. A. Belady, Program *evolution*, Academic Press, London and New York, 1985, pp. 99-122.

[Lehman85] Lehman M.M. and Belady L.A., *Program evolution*, Academic Press, London and New York, 1985.

[Lientz80] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*, Addison-Wesley, 1980.

[Linden95] F. J. van der Linden and J. K. Müller, Creating Architecture with Building Blocks, *IEEE Software*, Nov. 1995, pp. 51-60

[Martin83] Roger J. Martin and Wilma M. Osborne, *Guidance on Software Maintenance*, NBS Special Publication 500-106, Computer Science and Technology, U.S. Department of Commerce, National Bureau of Standards, p.6, Washington, DC, December 1983

[Parnas85] D. L. Parnas and P. C. Clements, D. M. Weiss, *The Modular Structure of Complex Systems*, IEEE Transactions on Software Engineering, March 1985, Vol. SE-11 No. 3, pp. 259-266, also published *in Proceeding of 7th International Conference on Software Engineering*, March 1984, pp. 408-417.

[Parnas94] Parnas D.L., Software Aging, *Proceeding of ICSE 16*, Sorento, Italy, pp.279-287, May 1994.

[Pearse95] Troy Pearse, Maintainability Measurement on Industrial Source Code Maintenance Activities, *Proceedings of International Conference on Software Maintenance 1995*, IEEE Computer Society Press, Opio (Nice), Oct. 17-20, 1995.

[Pesce95] M. Pesce, *VRML - Browsing and building cyberspace*, New Riders Publications, 1995.

[Rau97] A. Rau-Chaplin, B. MacKay-Lyons, T. Doucette, J. Gajewski, X. Hu and P. Spierenburgm, Graphics support for a World Wide Web based architectural design services, Computer Networks and ISDN Systems, Vol. 29, No. 14, October 1997, pp. 1611-1624.

[Rober91] G. G. Robertson, J. M. Mackinlay and S. K. Card, Cone Trees: Animated 3D Visualizations of Hierarchical

Information, *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '91)*, ACM Press, 1991, pp. 189-194.

[Roch93] J. B. Rochester and D. P. Douglas, *Re-engineering existing systems*, I/S Analyzer, Vol. 29, No. 10, Oct. 1991, pp. 1-12, reprinted in Arnold R., S. *Software Reengineering*, Computer Society Press, 1993, pp. 41-53.

[Santo97] H. P. Santo, Visualization and Graphics on the World Wide Web, Computer Networks and ISDN Systems, Vol. 29, No. 14, October 1997, pp. 1555-1558.

[Sarma96] A. Sarma, Introduction to SDL-92, *Computer Networks and ISDN Systems*, Elservier Science Publishers, Vol. 28, No. 12, 1996, pp. 1602-1615.

[Schach96] S. R. Schach, *Classical and Object-Oriented Software Engineering*, Irwin, 1996

[Shaw89] M. Shaw, Large Scale Systems Require Higher-Level Abstractions, *5th International Workshop on Software Specification and Design*, Pittsburgh, Pennsylvania, ACM, IEEE, May 19-20, 1989, pp. 143-146.

[Skram92] T. Skramstad and K. Khan, A Redefined Software Life Cycle Model for Improved Maintenance, *Proceedings of 1992 Conference on Software Maintenance*, Nov., 1992.

[Stasko92] John T. Statsko, *Three-Dimensional Computation Visualization*, Technical Report GIT-GVU-92-20, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, 1992.

[Swan76] E. B. Swanson, The dimension of maintenance, *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, IEEE Society Press, pp.492-497, October 1976.

[Swan80] E. B. Swanson and B. Lientz, *Software Maintenance: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison-Wesley Publishing Co., Reading, MA, 1980

[Tamai92] T. Tamai and Y. Torimitsu, Software Lifetime and its Evolution Process over Generations, *Proceedings of 1992 Conference on Software Maintenance*, Nov., 1992.

[Tufte83] Edward R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Connecticut, 1983.

[Turski96] W. M. Turski, Reference Model for Smooth Growth of Software Systems, *IEEE Transactions on Software Engineering*, Vol. 22, No. 8, Aug. 1996, pp. 599-600.

[Ulrich90] W. M. Ulrich, The evolutionary Growth of Software Reengineering and the Decade Ahead, *American Programmer*, Vol.3, No. 10, Oct. 1990, pp.14-20, reprinted in Arnold R., S. *Software Reengineering*, Computer Society Press, 1993, pp. 34-40.

[VRML97] The Virtual Reality Modeling Language, (VRML), ISO/IEC DIS 14772-1, 4 April 1997.

APPENDIX A

RIASSUNTO IN ITALIANO

La presente tesi di laurea si colloca nel contesto del progetto europeo ARES (Architectural Reasoning for Embedded Systems, ESPRIT Project #20477) che persegue l'obiettivo di studiare tecnologie innovative che servono agli sviluppatori software per descrivere, valutare e gestire architetture di famiglie di sistemi software. Nel perseguire tal fine, il progetto seleziona, estende o sviluppa un insieme di metodi, processi o prototipi software che permettono di ragionare sull'architettura di una famiglia di sistemi software durante il loro ciclo di vita.

Il presente lavoro è stato portato a termine presso il gruppo dei sistemi distribuiti dell'università tecnica di Vienna. Il gruppo e' diretto dal Prof. Mehdi Jazayeri il quale ha seguito lo svolgimento dell'intero lavoro.

Capitolo 1 Introduzione

I sistemi software evolvono per soddisfare le nuove richieste degli utenti o per dover adattarsi ai nuovi scenari a cui vengono sottoposti. L'evoluzione avviene modificando, correggendo e migliorando i sistemi stessi. Un punto critico viene raggiunto quando la complessità e la dimensione del sistema rendono ardue le nuove modifiche o gli sviluppi futuri: i nuovi prodotti richiedono più tempo per essere sviluppati e le modifiche richiedono esorbitanti costi. Questo fenomeno è indicato nella letteratura come invecchiamento del software. Per evitare un tale situazione le tecnologie note come reverse engineering offrono un aiuto per gestire i sistemi software esistenti.

Un precedente lavoro di tesi [Gursch96] ha proposto una metodologia per esaminare la struttura di un sistema software. La metodologia si basa sull'esame dell'evoluzione storica della struttura del sistema software. Tale esame è in grado di identificare eventuali difetti strutturali o identificare i componenti del sistema che richiedono una ristrutturazione. L'esame è stato eseguito su una base statistica a causa della vastità dei dati a disposizione. La quantità dei dati disponibili pone seri problemi su come estrarre informazioni utili. La possibilità di estrarre informazioni utili e' la chiave per poter incrementare la comprensibilità del sistema.

Lo scopo della presente tesi è quello di sviluppare una rappresentazione visuale che possa supportare l'esame della storia delle release software. La visualizzazione è una tecnologia emergente che permette di estrarre informazione da insiemi di dati estesi e complessi. È un intento di questa tesi mostrare che una rappresentazione visuale può aiutare a estrarre informazioni utili. Non è un intento della tesi quello di interpretare le informazioni estratte.

Capitolo 2 Conoscenza di base

Un sistema software, dopo essere stato consegnato all'utente, entra dell'ultima fase del suo ciclo di vita: la fase di manutenzione. In questa fase il sistema o i suoi componenti vengono modificati al fine di correggere errori, migliorare le prestazioni o gli attributi, o adattarsi ad un ambiente in cambiamento. Molte sono le ragioni che portano gli sviluppatori a modificare un prodotto esistenze. Il processo di modifica viene classificato in quattro tipi di lavoro: manutenzione correttiva per correggere gli errori trovati, manutenzione di adattamento per reagire all'ambiente in cui il prodotto opera, manutenzione di perfezionamento per soddisfare le richieste degli utenti e manutenzione preventiva per anticipare i futuri cambiamenti. I lavori di Belady e Lehman [Lehman76] [Lehman85] hanno suggerito che la fase di manutenzione non deve essere separata dalle altri fasi di sviluppo del software, ma sviluppo e successiva manutenzione devono essere considerate attività strettamente correlate e parti di un unico processo, chiamato processo di evoluzione del software. Un sistema software evolve, versione dopo versione, aggiungendo nuove funzionalità, rimuovendone altre o cambiando la sua struttura. L'evoluzione e' un aspetto intrinseco di un sistema software e perdura finché il prodotto non viene abbandonato. Però le continue modifiche possono alterare l'integrità strutturale del sistema ad un punto tale che successive estensioni diventano difficili, richiedono molto tempo, interessano molti componenti del sistema o in generale diventano più costose. Questo fenomeno viene chiamato invecchiamento del software. La struttura può degradare ad un punto tale che può terminare il ciclo di vita del sistema. Per evitare un tale invecchiamento sono state sviluppate una serie di tecnologie che sono raggruppate sotto il nome di reverse engineering. Il loro scopo è quello di esaminare e gestire un sistema software esistente in modo da incrementarne la comprensibilità generale.

Capitolo 3 Il caso di studio

Il caso di studio è il sistema software di una famiglia di switching telefonici. Il sistema è stato sviluppato all'inizio degli anni '80 e poi e' stato sottoposto ad una serie di successive estensioni e ristrutturazioni per adattarlo a nuovi scenari. Il sistema è in continua evoluzione e ora (1998) ha raggiunto una dimensione di tredici milioni di linee di codice.

L'architettura del sistema è a quattro livelli: sistema, sottosistema, modulo e programma. Ogni livello provvede delle funzionalità al livello superiore. Ciascun livello è costituito da un certo numero di elementi: sistemi, sottosistemi, moduli e programmi. I programmi sono il livello più basso del modello astratto e sono associati ad un certo numero di file sorgenti. Ad ogni versione del sistema

(chiamata release) uno o più programmi vengono modificati per implementare le nuove funzionalità. Un sistema di numerazione viene adottato per tener traccia delle differenti implementazioni dei programmi. A ciascuna implementazione viene associato un numero progressivo di versione che permette di identificarla univocamente. Anche le release del sistema sono numerate progressivamente. I sistemi di numerazione sono tutti indipendenti. In questo modo dato un numero di release del sistema è possibile individuare la configurazione dei programmi che vengono usati per l'implementazione. Le informazioni sui numeri di versione dei programmi e sui numeri di release sono immagazzinate in una base di dati. Il caso di studio e' costituito da venti release di sistema e da circa 1500-2300 programmi.

Capitolo 4 L'analisi della storia delle release software

L'analisi della storia delle release software è un metodo per valutare la struttura di un sistema software basandosi sull'uso dell'informazione storica. L'obbiettivo è quello di scoprire eventuali difetti o di identificare i moduli che richiedono ristrutturazione. Il metodo assume che il sistema sia decomposto in moduli e un sistema di numerazione sia usato per tener traccia delle versioni dei moduli, come per il caso di studio. Esaminando queste informazioni è possibile estrarre delle considerazioni sull'evoluzione del sistema.

Capitolo 5 Stato dell'arte

L'esame della storia delle release software può essere coadiuvato dall'uso di due tool che sono in grado di visualizzare l'informazione di un sistema software. Il primo si chiama Navigator ed è stato espressamente sviluppato per visualizzare graficamente la struttura di un sistema e facilitarne la navigazione. Il secondo si chiama SeeSys e permette di visualizzare statistiche associate al codice sorgente di un sistema software. I due sistemi sono comparati sulla base di alcuni fattori che possono essere vantaggiosi quando si esamina la storia delle release software.

Capitolo 6 Descrizione del problema

La storia delle release software è costituita da tre entità: tempo, struttura e attributi. La dimensione temporale è misurata in tempi di release. Ad ogni tempo di release è associato una versione del sistema software. La struttura di una generica versione software è ottenuta decomponendo il sistema in moduli e relazioni tra questi. Ad ogni modulo viene associato una parte del codice sorgente del sistema software. Per ogni modulo di una release del sistema gli attributi, come numero di versione, dimensione e complessità, vengono calcolati basandosi sul codice sorgente del modulo stesso. L'esame di queste informazioni deve produrre delle osservazioni sull'evoluzione del sistema stesso.

La quantità di dati presenti nella storia delle release software porta alla luce il problema di come estrarre l'informazione utile per esaminare il sistema. Il precedente lavoro di Gurschler [Gursch96] ha eseguito un'analisi statistica e si è basato su una diretta ispezione dei dati. Lo scopo della presente tesi è quello

di investigare la possibilità di visualizzare l'informazione contenuta nella storia delle release software. È un intento di questa tesi mostrare che una rappresentazione visuale dei dati può aiutarne l'esame.

La visualizzazione delle informazioni è una tecnologia emergente che si propone il fine di facilitare la comprensione di complessi e estesi insiemi di dati. I principali vantaggi che si possono ottenere sono: un approccio percettivo alla comprensione dell'informazione, possibilità di individuare sequenze ricorrenti di dati e efficacia nel creare l'interesse dell'osservatore.

Capitolo 7 Visualizzazione della storia delle release software

L'approccio adottato in questa tesi, per sviluppare una rappresentazione visuale che sia di aiuto all'esame della storia delle release software, si basa sull'uso di diagrammi a tre dimensioni. In un diagramma a tre dimensioni vengono visualizzate contemporaneamente le tre entità che costituiscono la storia delle release: tempo, struttura e attributi. La struttura viene visualizzata tramite diagrammi a due o tre dimensioni. L'evoluzione temporale fa uso della terza dimensione. I valori degli attributi vengono visualizzati tramite differenti colori.

L'uso della rappresentazione visuale è limitato dai dati forniti dal caso di studio, ossia i numeri di versione dei programmi. La rappresentazione visuale di questa informazione è stata usata per l'esame del caso di studio.

I vantaggi principali dell'approccio usato sono: visualizzazione simultanea delle tre entità che fanno parte della storia delle release software, l'uso del colore e della terza dimensione, l'uso di meccanismi per ridurre l'informazione visualizzata, la possibilità di visualizzare l'evoluzione storica e la possibilità di navigare la struttura del sistema.

Capitolo 8 Sistema di visualizzazione 3DSoftVis

Un tool è stato sviluppato per supportare la rappresentazione visuale sviluppata in questa tesi. Il tool è realizzato in Java ed utilizza due componenti: un browser per il web e un browser per VRML. L'applicazione supporta la visualizzazione tridimensionale dei grafici, l'uso di più finestre, permette di impostare i dati da visualizzare e può essere utilizzata tramite la rete.

Capitolo 9 Conclusioni

Il presente lavoro di tesi ha prodotto i seguenti maggiori contributi alla conoscenza: sviluppo di una rappresentazione visuale per supportare l'esame della storia delle release software, l'uso della terza dimensione e del colore sono stati investigati come mezzi per aggiungere informazione ad una rappresentazione visuale, la visualizzazione ha dimostrato la sua utilità per comprendere complessi ed estesi insiemi di dati, un supporto software per la visualizzazione è stato sviluppato.

APPENDIX B

VISUALIZING HIERARCHIES

This appendix describes the mathematical concepts used in this thesis for visualizing graphically hierarchies. A hierarchy is visually represented by a tree structure. Two types of tree structure has been used: 2-D Tree and 3-D Tree. This section explains how they have been implemented.

2-D Tree

2-D Tree displays a tree structure using two dimensions. The width and the height of the tree are calculated using a recursive formula. Considering Fig. 1 d_{n-1} is the width of the father element (indicated with E_{n-1} in the picture) and d_n is the width of a child element (indicated with E_n in the picture). Each child element has the same width d_n. The height of the tree is the distance between the father element and its children, it is indicated with h_{n-1}.

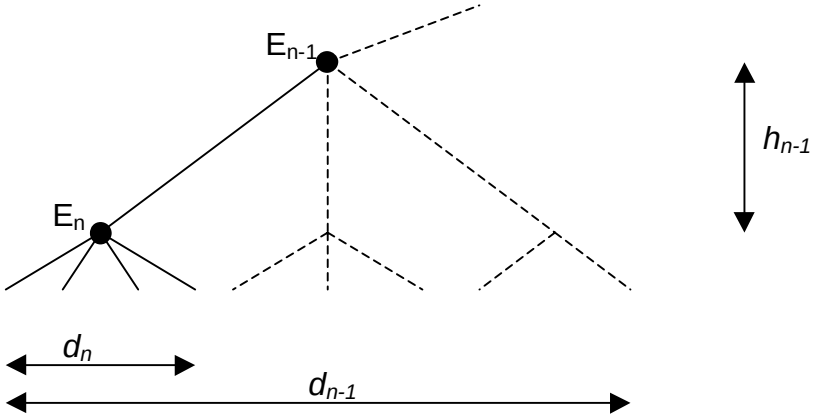

Fig. 1 2-D Tree layout.

The values of width and height are calculated using the following formula:

$$d_n = \frac{d_{n-1}}{N_{n-1}}$$

$$h_n = h_0$$

where N_{n-1} is the number of children of the father E_{n-1}, h_0 is a constant. The starting value d_0 is assigned by the user.

3-D Tree

3-D Tree displays a tree structure using the technology of ConeTree [Rober91]. ConeTree makes it possible to display large number of nodes by using 3D space. Fig. 2 shows the three dimensional layout. Two levels are shown: level *n-1* and level *n*. Level *n-1* represents the father element, level *n* represents the child elements. Each element of the tree is displayed using a cone. Cones are visualized in the picture as circles. The father element displays its child elements in the space of its cone. The father's space is partitioned in slices and each slice is assigned to a child element. The child elements use the assigned space to display their own cone.

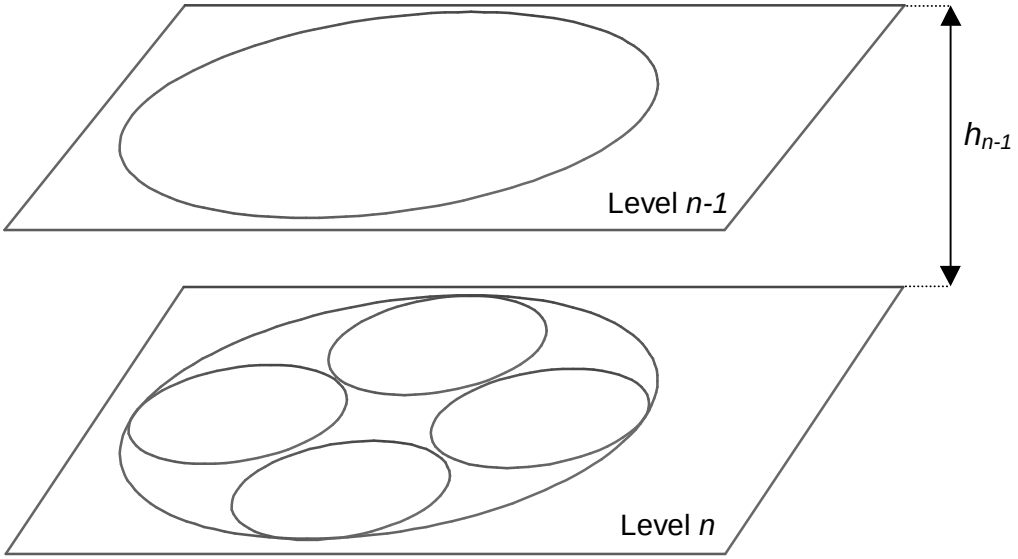

Fig. 2 3-D Tree layout.

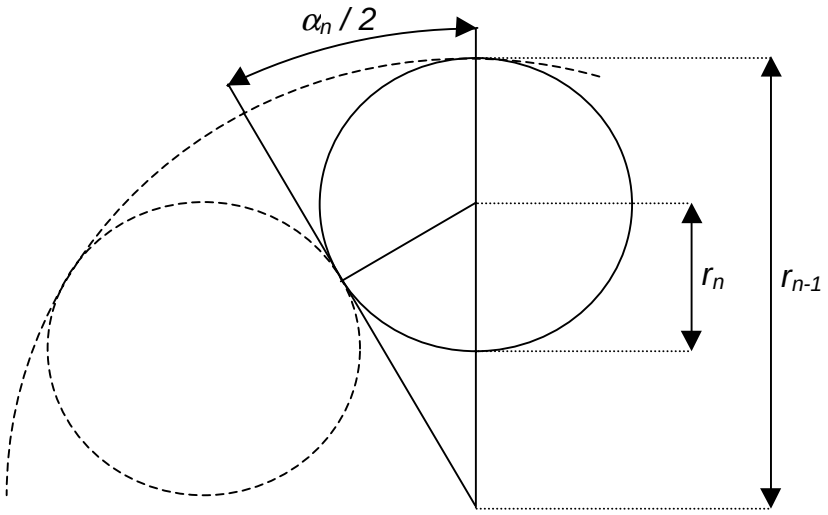

Fig. 3 Top view of a cone tree.

Fig. 3 represents the top view of the elements at level *n*. It shows both the father's cone and the children's cone. Indicating with r_{n-1} the radius of the father's cone, with r_n the radius of the children's cone and with N_n the number of children at level *n* the recursive formula for calculating the radius is:

$$\alpha_n = \frac{2 \cdot \pi}{N_n}$$

$$r_n = \frac{r_{n-1}}{1 + \frac{1}{sin\frac{\alpha_n}{2}}}$$

$$h_n = h_0$$

where h_n is the distance between level *n* and level *n-1*, h_0 is a constant value. The starting value r_0 is assigned by the user.