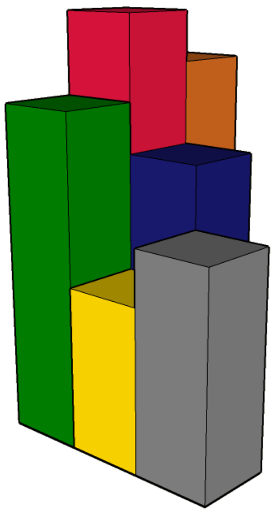
Development of a 3D tool for visualization of different software artifacts and their relationships

David Montaño Ramírez



Development of a 3D tool for visualization of different software artifacts and their relationships

Master Thesis

David Montaño Ramírez
Advisor: Jairo Hernán Aponte Melo

Universidad Nacional de Colombia
Departamento de Ingeniería de Sistemas e Industrial
2010

To my mother Mercedes, my father José, my grandparents Ana and Misael and to my girlfriend Yury

Acknowledgments

This thesis would not be possible without my family and the support they gave me. Thanks to Professor Jario Aponte for his invaluable help and guidelines. I want to express my gratitude to Professor Andrian Marcus for dedicating time to support this work. I also want to thank to my girlfriend Yury Niño for helping me through this process. Thanks to the Colswe research group for their time reviewing this work. Special thanks to all the people who help me and help others in the discussion forums in the Internet, they provide me great information resources when I needed them. Finally thanks to everyone involved in the development of this work.

Contents

1 Introduction **1**

2 Background **4**

2.1 Software Evolution 4

2.2 Software Visualization 6

2.2.1 Metaphors 7

2.2.2 Visualization Overview 8

2.2.2.1 Virtual Environments (VE) 9

2.3 Visualization Sources and Techniques 9

2.3.1 Static Source 9

2.3.1.1 Representing Code with Text 10

2.3.1.2 Representing Code with Diagrams 10

2.3.1.3 Visualizing Software Architectures 11

2.3.2 Dynamic Source 12

2.3.2.1 How data is collected? 12

2.3.2.2 What data is collected? 12

2.3.2.3 Fundamental Techniques 13

2.3.3 Evolutionary Information Source 13

2.3.3.1 Software Metrics through Time 14

2.3.3.2 Visualization of Structural Change 14

2.3.3.3 Visualization of Evolutionary Coupling 15

2.4 UML and Software Visualization 16

2.5 What has been done? 16

2.6 Summary 18

3 SeeIT 3D 19

3.1 The Metaphor . . . 19

3.1.1 Relationships among Containers . . . 22

3.2 Underlying Technologies . . . 23

3.2.1 The IDE . . . 24

3.2.2 The Graphics Engine . . . 24

3.2.2.1 JOGL . . . 25

3.2.2.2 Scene Graph Based Engines . . . 25

3.2.2.3 Choosing the Graphics Engine . . . 26

3.3 Internal Construction . . . 27

3.3.1 Metrics . . . 28

3.4 XML Based Visualization . . . 29

3.5 Color Scales . . . 30

3.6 Summary . . . 31

4 Usability Concerns on SeeIT 3D 32

4.1 IDE Interactions . . . 32

4.2 Visualization Interactions . . . 34

4.3 Summary . . . 37

5 SeeIT 3D to Understand Open Source Software Projects 38

5.1 aTunes . . . 39

5.1.1 Visual Analysis . . . 39

5.2 DrJava . . . 39

5.2.1 Visual Analysis . . . 39

5.3 JFreechart . . . 42

5.3.1 Visual Analysis . . . 43

5.4 Spring Core . . . 46

5.4.1 Visual Analysis . . . 46

5.5 Wicket . . . 47

5.5.1 Analysis . . . 47

5.6 Comparing five projects . . . 49

5.7 Visualizing a Relational Database . . . 51

5.8 General Conclusions . . . 53

5.9 Summary . . . 54

6 Tool Evaluation **55**

6.1 Comparing with Other Similar Tools 55

6.1.1 X-Ray 56

6.1.2 Codstruction 56

6.1.3 Comparison 58

6.2 Evaluation Framework 59

6.2.1 Framework Description 61

6.2.2 Evaluation 62

6.2.2.1 Intend 62

6.2.2.2 Information 62

6.2.2.3 Presentation 63

6.2.2.4 Interaction 63

6.2.2.5 Infrastructure 64

6.2.2.6 Support 64

6.2.3 Evaluation Results 64

6.3 Summary 65

7 Conclusions and Future work **66**

A Information about the SeeIT 3D Development **68**

A.1 Research Production 68

A.2 Additional Resources 68

B XML Example **69**

List of Figures

2.1 Structograms . . . 11
2.2 Jackson Diagrams Representation . . . 11
2.3 Control Structure Basic Diagrams . . . 11
2.4 X-Tango Visualization . . . 13
2.5 JSamba in action . . . 14
2.6 Bubble sorting using a three dimensional space . . . 14
2.7 SeeSoft Example . . . 15
2.8 Structural change through time . . . 15
2.9 Support graph of Mozilla Firefox . . . 16

3.1 SeeIT 3D metaphor elements . . . 21
3.2 Bidimensional relationship example . . . 22
3.3 Visualizing relationships by using *Common Base* . . . 22
3.4 Visualizing relationships by using *Lines* . . . 23
3.5 Visualizing relationships by using *Arcs* . . . 23
3.6 Visualizing relationships based on movement . . . 24
3.7 Google trends results about downloads of the most popular IDEs. Blue: Eclipse, Red: Visual Studio, Orange: NetBeans, Green: JDeveloper, Purple: JBuilder . . . 25
3.8 SeeIT 3D internal architecture . . . 27
3.9 Information structure of a XML based visualization file . . . 30

4.1 SeeIT 3D general view . . . 33
4.2 Visualization availability within the IDE . . . 33
4.3 Scaling containers . . . 35
4.4 Package explorer vs. Visualization Synchronization . . . 35

4.5 Different granularity levels 35
4.6 SeeIT 3D main interaction view 36
4.7 Drag and drop based mapping 36
4.8 Information selection options 37

5.1 aTunes overview at package level 40
5.2 aTunes overview at class level 40
5.3 Relationships among aTunes artifacts 41
5.4 DrJava overview at package level 41
5.5 DrJava overview at class level 42
5.6 Relationships among DrJava artifacts 43
5.7 JFreechart overview at package level 44
5.8 JFreechart overview at class level 44
5.9 Relationships among JFreechart artifacts 45
5.10 JFreechart highest complexity artifact 45
5.11 Spring core overview at package level 46
5.12 Spring core overview at class level 47
5.13 Relationships among Spring core artifacts 48
5.14 Wicket overview at package level 48
5.15 Wicket overview at class level 49
5.16 Relationships among Wicket artifacts (using arcs) 50
5.17 Relationships among Wicket artifacts (using common base) 50
5.18 Side by side overview of five projects 51
5.19 Side by side five project mapping changed 52
5.20 Relational database example 52
5.21 Relational database viewed with SeeIT 3D 53

6.1 X-Ray metaphor concept[27] 56
6.2 Class and Package dependency view 57
6.3 JFreechart visualization with X-Ray 58
6.4 JFreechart visualization with Codstruction 59

List of Tables

2.1 Challenges in Software Evolution 4

2.2 Non-Exhaustive list of Applications 16

3.1 Metaphor use. Concrete example 21

3.2 Metrics used in SeeIT 3D 29

3.3 Color scales used in SeeIT 3D 31

6.1 SeeIT 3D Vs. X-Ray Vs. Codstruction 60

Chapter 1

Introduction

The great impact that software systems have in our lives, makes the study of them an important task that has to be carried on carefully; this study includes maintenance, enhancement, correction and understanding tasks. These systems are abstract entities that do not have a natural representation to our senses. This makes them difficult to understand as a whole, where the components interact with each other with a common purpose. The lack of representation of software systems requires developers to use visualization mechanisms for several purposes, such as understanding them, detecting security problems, or finding bugs and ways to improve them.

Software systems evolve through time according to the *software evolution* theory. This evolution becomes important when a system is desired to last through time. The research on software visualization is immersed in the software evolution field where it can be used to support development tasks on the topics specified by this wider research area, such as understanding and maintenance. These tasks enable software visualization to become an important actor in the evolution process.

Software visualization aims to be a tool to face the challenges proposed by software evolution, using visual techniques to provide several types of views humans can understand and analyze the apparent complexity of the existing software [19]. The main challenge is to find effective mapping among different aspects of software to graphical representations by using visual metaphors [19]. In 2005 a survey [7] showed that software visualization is considered a powerful tool when performing development tasks. In this survey, 40% of researchers considered software visualization absolutely necessary for their work, while 42% considered it important but not critical.

Software visualization has a long history, which started when it was first used in algorithm animation [2], and goes until the usage of virtual environments [25, 19], recently developed. Various evaluation frameworks have been developed to evaluate and compare software visualization tools; in addition, they can help developers to assess the suitability of a tool for specific maintenance tasks. One of the first who propose an evaluation framework was Price *et al.* in [34]; they included key aspects to be considered when building a software visualization tool, such as: scope, content, form, method, interaction and effectiveness.

Currently, the software visualization process is based on three main sources of information: *static, dynamic* and *evolutionary information*. *Static* refers to information that can be extracted from software artifacts that do not need the program to be running, these include: source code, documents, diagrams and related artifacts. *Dynamic* refers to information extracted from the execution environment of the program; this means that variables values, execution stack and statements executed are part of this kind of information. Finally, *evolutionary* information refers to the information that is extracted[1] from the evolution history of the analyzed system.

Despite of the efforts made by researchers, developers have not found a tool powerful and intuitive enough to fulfill their needs. This fact is highlighted by the lack of impact of the visualization tools in development teams. For example, in the development of Eclipse (IDE) plugins, there are many of them in Eclipse Marketplace[2], but none has had a notable impact as the IDE itself, which can be considered as the facto IDE for Java development. But nowadays, when computer processing level has increased dramatically and software systems are more involved in our daily life than ever, is mandatory to develop means to understand these systems with simpler methods than just browsing and reading the source code trying to figure out how and why it works in particular way.

With this in mind, developers should be able to interact with a tool that provides a view of the system that can be used as a maintenance instrument. The tool is encouraged to face the visualization problem from two sides: first to consider different software artifacts generated by the development process, and second to integrate the visualization tool within an IDE. The first aspect should take into account the source code (with its different granularity levels), related documents, evolutionary history and any other artifact that can be analyzed. Because of this last requirement, the visualization tool must have a flexible metaphor as well as an input data mechanism able to manipulate different kinds of information. The second aspect will facilitate the interaction with the visualization tool, by avoiding the time-consuming task of synchronizing the IDE with the visualization tool when artifacts are changed. Also, considering the architecture of the current IDEs, it is possible to integrate a wide variety of tools to help the understanding of the system. This is the case of Eclipse, IntelliJ, Netbeans, JDeveloper and JBuilder to name a few.

Visualizing relationships among software artifacts has been widely used in two dimensional metaphors. These metaphors have used graph-like structures i.e. lines to connect a set of related nodes. On the other hand, not so many three dimensional based metaphors have included work done in this area. Techniques proposed within two dimensional approaches for this purpose, can be used as a starting point for developing relationship based metaphors in a three dimensional world.

The visualization process is not closed to the metaphor definition; they also need to consider the usability principles in order to be fully compliant with users' requirements. When using a three dimensional visualization tool, these principles need to be carefully analyzed because the user can get lost easier than in the two dimensional model. Even more, the use of a metaphor with high flexibility level makes necessary to provide the user with a way to select the mapping between visual properties and the characteristics of the artifact itself.

As long as the software visualization systems are getting better, there will be an increasing interest from developer's point of view in exploring and understanding the software system. This interest requires the definition of evaluation frameworks that consider important aspects such as the sources of information, end user classification and mediums of representation. The focus in this area should be on the development of metrics to support the evaluation process. Previously developed frameworks have only proposed qualitative evaluations as it was shown in [32], so there is a clear need in this area.

[1]The information extraction is a complete research field, for the purpose of this document the extraction process is omitted
[2]marketplace.eclipse.org/

This document presents the process done in the development of SeeIT 3D, a three dimensional visualization tool integrated within an IDE. The next chapter presents the background of the software visualization area as well as the most common visualization techniques applied in the field. In chapter 3 the foundations of SeeIT 3D as well as the underlying technologies are presented. On chapter 4 because of the importance of usability in the visualization pipeline, the concerns about it are presented. Then on chapter 5, SeeIT 3D is presented as a tool to understand open source projects; it shows how the tool can help developers to understand the system and to find code smells. Finally the conclusions highlight the work done, how it may contribute to the area and why it is useful to continue the work in the field.

Chapter 2

Background

The software visualization process is immersed within software evolution; as such it must face the challenges of this field. This chapter presents these challenges and how the software visualization techniques have contributed to the research by supporting a set of tasks that must be done in the software evolution area.

2.1 Software Evolution

Software evolution is a concept that should be placed in the core of development process. By embracing this concept, along with its principles and practices, is possible that the life cycle of software systems can be prolonged over the time. It also allows developers to update essential features of software systems according to a world that naturally evolves. This means that if change is not considered as a vital process in software development, sooner or later the system will become quickly obsolete and useless.

Back in 2005 a list of challenges [30], enunciated in table 2.1, were established and they should be addressed in order to embrace the software evolution principles and goals as part of the software production models. These challenges established a path in the research field of software evolution as they cover a wide area of it, by integrating different aspects in software development and maintenance.

Table 2.1: Challenges in Software Evolution

#	Challenge	Description
1	Preserving and improving software quality	To provide tools and techniques that preserve or even improve the quality characteristics of a software system, whatever its size and complexity
2	A common software evolution platform	To develop and support a common application framework for doing joint software evolution research

3	Supporting model evolution	Software evolution techniques should be raised to a higher level of abstraction, in order to accommodate not only evolution of programs, but also evolution of higher-level artifacts such as analysis and design models, software architectures, requirement specifications and so on
4	Supporting co-evolution	The necessity to achieve co-evolution between different types of software artifacts or different representations of them
5	Formal support for evolution	In order to become accepted as practical tools for software developers, formal methods need to embrace change and evolution as an essential fact of life
6	Evolution as a language construct	Programming (or even modeling) languages should provide a more direct and explicit support for software evolution
7	Support for multi-language systems	Must provide more and better support for multi-language systems
8	Integrating change in the software life-cycle	It is important to investigate how the notion of software change can be integrated into the conventional software development process models
9	Increasing managerial awareness	Increase awareness of executives and project managers of the importance and inevitability of software evolution
10	Need for better versioning systems	To develop new ways of recording the evolution of software that overcome the shortcomings of the current state-of-the-art tools
11	Integrating data from various sources	To find out how these different kinds of data can be integrated, and how support for this integration can be provided
12	Analyzing huge amounts of data	New techniques and tools are needed to facilitate manipulation of large quantities of data in a timely manner
13	Empirical Research	Need for more empirical research to measure impact of: process models, tools, languages, people

14	Need for improved predictive models	Predictive models are crucial for manager in order to assess the software evolution process. These models are needed for predicting a variety of things: where the software evolves, how it will evolve, the effort and time that is required to make a change
15	Evolution benchmark	To come up with, and reach a consensus on, a common set of evolution benchmarks and case studies which, together, are representative for the kinds of problems needing to be studied
16	Teaching software evolution	How to integrate ideas, formalism, techniques and tools for software evolution into our computer science curriculum in a meaningful way
17	A theory of software evolution	It is necessary to develop new theories and mathematical models to increase understanding of software evolution, and to invest in research that tries to bridge the gap between the what of software evolution and the how of software evolution
18	Post-deployment runtime evolution	Need for proper support if runtime adoptions of systems while they are running, without the need to pause them, or ever shut them down

Considering these challenges, software visualization systems are able to contribute to seven of them. First, preserving and improving software quality by allowing to understand and detect errors easier than analyzing the source code; second, supporting model evolution by visualizing different software artifacts allowing developers to embrace the underlying model in a more practical way. Also providing different views (like filtering mechanisms) that promote better modeling of the system; third, support for multi-language systems by using a metaphor independent from the source of information; fourth, increasing managerial awareness by exposing views easily understandable by managers and others involved; fifth, integrating data from various sources by providing a more complete view of the system; sixth analyzing huge amounts of data by using 3D systems where it is possible to analyze more information; seventh, teaching software evolution by making easier to highlight concepts when they are seen by students.

2.2 Software Visualization

Software visualization systems are considered useful tools in software development because they provide a method to understand and represent the overwhelming amount of information produced by analysis tools. Because of their usefulness and importance, they need to be built upon a set of dimensions as stated by Maletic *et al.* in [26], these include:

- *Tasks*: This dimension answers the question: Why the visualization is needed?. It specifies what particular software engineering task is supported by the software visualization system. These tasks include: development activities (e.g., programming, debugging, testing, etc.), maintenance (e.g., fault detection, re-engineering, reverse engineering, etc.) and even educational tasks.
- *Audience*: This dimension answers the question: Who will use the visualization?. The audience dimension defines the attributes of the users of the visualization system. These include students and instructors in educational environments and developers, designers, testers, etc., in industrial environments.
- *Target*: This dimension answers the question: What is the data source of the visualization?. It defines what (low level) aspects of the software are visualized. The target is a work product, artifact, or part of the environment of the software system. Examples of targets are architectures, designs, algorithms, source code, execution/trace information, measurements and metrics and documentation.
- *Representation*: This dimension answers the question: How it is represented?. It defines how the visualization is constructed based on the available information. The representation manifests itself as the visual structures in the reference model. It uses the defined metaphor to map the analyzed information of the target to a visual property available.
- *Medium*: This dimension answers the question: Where the represented data will be visualized?. It is where the visualization is rendered (e.g. paper or screens)

By considering these dimensions, developers of software visualization systems are following the path that leads to a system that assists the process of software development.

2.2.1 Metaphors

One of the fundamental concepts behind any kind of visualization is the *metaphor*. It was defined by Lakoff [20] as “a rhetorical figure whose essence is the understanding and experiencing one kind of thing in terms of another”. In software visualization, metaphors are the most important concern because the information that is going to be visualized does not have a natural visual representation.

Since the first appearance of software visualization in algorithm animation [2], metaphors have been developed using different techniques such as bar charts, pie charts, cylinders, pixel-maps, buildings within cities and even galaxies in the universe.

When building metaphors designers must consider a set of basic aspects, as defined by Gračanin [12], before they can be included in a visualization system; these aspects include:

- Scope of representation: Software systems usually consist of thousands of lines of code and the visualization tool has to render information related with them. This vast amount of information often causes confusion to the end user. Any metaphor should allow the user to limit the scope of the information that is being visualized, so he can decide what information is relevant or not.
- Medium of representation: One kind of medium is 2D or 3D visualization type. The medium has an important role when building a software visualization system as it is usually attached to the kind of information that is being visualized.

- Visual metaphor: This aspect refers to what visual elements the metaphor uses to display information to the user. This includes geometric shapes such as: lines, dots, circles, squares and polygons, or real-world entities such as buildings, trees, planets and so on. These elements may have a color (in a color scale) to represent another aspect of a software artifact. Considering these elements and how they are used, a metaphor needs to define:
  - *Consistency of the metaphor:* It refers to the correct use of the metaphor. This means that it must exist a mapping between different entities in software and entities in the visualization to avoid confusing the user with the representation of different objects against the same property in the visualization.
  - *Semantic richness of the metaphor and complexity*: The metaphor should be rich enough to provide as much representations as different aspects of the software that is being visualized.
- Abstractedness (Ellison): The user of the visualization system should be able to choose the level of detail in the software system that is being evaluated. This way the user may choose from direct representation, structural representation, synthesized representation, and analytical representation.
- Ease of navigation and interaction: Since the visualization system is going to provide too much information, it should allow the user to know what information is visualizing, where in the system he is, what level of abstraction has selected and it should allow him to navigate in an understandable way according to some usability criteria. This is an important aspect in 3D visualizations where the user can easily get lost.
- Level of automation: Software visualization systems need to be automated, i.e. extract, analyze and render all the information from a software system with a minimum interaction from the user.
- Effectiveness [24]: It indicates the efficacy of the metaphor as a medium of representing the information. The metaphor should be able to transmit the analyzed information from the software system, for example if it is able to show a numeric value as well as a cardinal value.
- Expressiveness [24]: It refers to the capability of the metaphor to visually represent all the analyzed data. As in the case of semantic richness, the metaphor must provide a considerable number of visual properties so the parameters obtained by the analysis can be represented in the view.

2.2.2 Visualization Overview

Different kinds of techniques are involved when visualizing information. In software visualization, bidimensional graphics have been widely used applying different techniques dominated by tree-like and graph representations built from geometrical shapes [12]. They often consist of several thousands of nodes and arcs (due to the complexity of a software system). But research has not stopped there, nontraditional techniques like treemaps, pixel-maps and Fractals [43, 9] has been developed to show several kinds of information.

In the development of 2D based metaphors, researchers have found a major problem with the amount of information presented to the user because of the complexity and size of the analyzed software. This huge amount of information confuses the user instead of providing a wider knowledge about the system. Stasko in [37] states about visualizing in a three-dimensional world, "by adding an extra spatial dimension, we supply visualization designers with one more possibility for describing some aspect of a program or system", thus more information is easily represented. When using a 3D metaphor, it also has been suggested that the perception is less error prone if software objects are mapped to visual objects, as there is a natural mapping between them [41].

2.2.2.1 Virtual Environments (VE)

Virtual environments give the user a unique type of immersion and navigation because of the way they represent and render the information. This level of interaction is achieved by presenting to the user a world where he is able to *interact* with different objects that are mapped to software components/artifacts.

Research in this area is far more complicated as it requires more human and technological resources. Therefore, the work in the area has not been as popular as 2D approaches. Despite of these costs, certain work has been done; this is the case of ImsoVision [25], which visualizes C++ source code. It employs geometrical figures to represent the different components in the software system. While this work have had impact on the C++ community, Software World [19] has done it for the Java language; it makes use of a metaphor based on elements from real world like countries, districts and buildings, to represent the source code. Additionally, one of the most important visualization tools is CAVE[1] proposed on [5], which uses a cubicle where the user interacts with the world presented on it.

Distributed VEs are a special kind of virtual environment where many users, distributed in different places, interact with the visualization at the same time. By providing a tool capable to support this kind of operations, all users involved in the visualization can interact with each other, and work in a collaborative style.

2.3 Visualization Sources and Techniques

Visualization systems are generally based on software metrics, as they reflect some specific software property that can be visualized. IEEE standard 1601 defines them as "A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software has a given attribute that affects its quality". This way, a metric can be as simple as the number of lines of code in the program or as complex as the lack of cohesion metric. Visualization tools use color scales, sizes, geometrical shapes and other visual properties to represent the values of these metrics. They can be classified according to the source where they can be extracted. In the following sections the basic sources of information are presented, ranging from simple language statements to evolutionary information of the system.

2.3.1 Static Source

Static source of information is extracted from a system that is not needed to be running. Therefore no information from runtime can be extracted. The main source of *static* information is the source code. The

[1]Not only for software visualization

next subsections present techniques that deal with this kind of information, as well as diagrams based on the source code itself.

2.3.1.1 Representing Code with Text

The first type of static visualization is the source code itself. It is a representation from the compiled code interpreted by the computer processor. Textual representation of software is the most used because is the mechanism employed by developers to make software systems. Because of this importance, there has been great interest in making it better and easier to understand and maintain. Based on the work done by Diehl on [8] the techniques that process this kind of data are listed below.

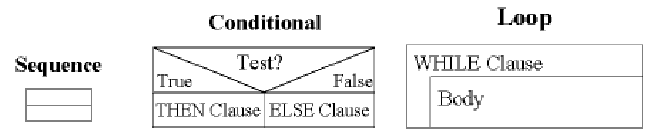
- Pretty printing: The goal of pretty printing is to make the nesting of blocks of code visible, while using the minimal number of lines for each block and allowing the programmer to see some kind of structure in the code. The first step toward the implementation of this representation was using blank spaces, indentation and line brakes. After some time, with the advantage of technologies, these techniques were complemented with font types, font color and different sizes. This was fine but did not provide enough feedback to the programmer to easily understand what the program was doing and how it collaborated with other parts of the system.
- Program as Publication: Back in 1984 Donald Knuth introduced the term literate programming, proposing that every program should be considered as a literature work. Years later Baecker and Marcus [1] proposed the use of program books. The system was documented with the structure of a book, including one chapter with the program documentation, and each source file as a set of pages with pretty printing styled format, separation of methods and comments on the side of the program code.

2.3.1.2 Representing Code with Diagrams

Another well known form of static visualization are Diagrams, they include control-flow, chart, structograms [31], Jackson [17] and Control Structure diagrams [14]. This type of representation provides the user an easy way to understand the program, because it is based on geometrical figures that the human brain processes and retains better.

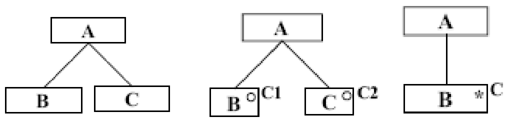
Control-flow diagrams: In 1947 Von Neumann [11] created one of the most famous ways to visualize the flow of a program by using geometrical figures to represent actions or events within the application. These actions were represented by *Rectangles* when the flow of the program referred to events, activities, processes, functions and other general statements, and by *Diamonds* when the flow got a point where a decision had to be taken. This is a simple yet powerful way to better explain and understand basic algorithms, for example sorting algorithms, but it falls down when the program gets bigger. For this reason, since then, researchers have developed tools to automatically generate diagrams with the proper layout and configuration.

Structograms: Searching for a way to write programs in a more structure way, Nassi-Shneiderman proposed a new way to diagram programs based on rectangles to represent them (see figure 2.1). Since it does not have a representation of the GOTO statement, the programmer is forced to write programs without it



(a) Statement (b) Conditional (c) Loop

Figure 2.1: Structograms



(a) Statement (b) Conditional (c) Loop

Figure 2.2: Jackson Diagrams Representation

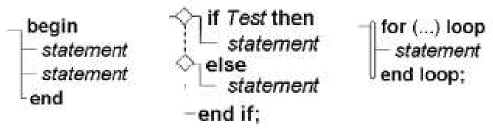
(when this representation was proposed, Object-Oriented Programming was not as popular as it is today), so making them more structured and easier to maintain and understand.

Jackson Diagrams: They represent a program as a tree hierarchy providing a format to depict the structure of the source code. Figure 2.2 has the diagrams for three control structures.

Control structure diagrams: These diagrams take the control-flow charts and the source code together. They assemble the diagram into the source code by showing on the side the figure that represents the statement. For example, if a line contains a conditional, it is marked with a diamond on the left side; similar with loops, where a vertical bar indicating all the block is placed on the left side as it is shown in figure 2.3.

2.3.1.3 Visualizing Software Architectures

Software architectures are a point of view of the system where the artifacts and components can be seen as they interact with each other. It is defined in the IEEE 1471 standard as: "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution". Types of software architectures are: Pipes and Filters, Layered Systems and Blackboards. These have been created to solve common programming problems and have been accepted by developers because they embody successful and maintainable solutions.



(a) Statement (b) Conditional (c) Loop

Figure 2.3: Control Structure Basic Diagrams

Software architectures need to be analyzed from their components in order to be visualized. This task makes difficult for a visualization tool to automatically guess the architecture based on its components and how they interact. This way, it is almost impossible to visualize the architecture as it should (i.e. using the correct metaphor for the correct architecture such as layers for a layered architecture).

The most successful approach in building a metaphor that allows to observe the complexity of a software architecture, is the Unified Modeling Language (UML) proposed by the Object Management Group (OMG) in 1997. Nowadays, it is widely used by software development groups, because of the easiness to understand and its capability to represent many aspects of the system under consideration. After the apparition of UML, an enhancement was proposed in [16, 15]. They used a 3D space to visualize the same elements of UML, but the success was not as important as UML due to the difficulty for drawing the shapes on a paper or a whiteboard.

2.3.2 Dynamic Source

The visualization that depends on the data extracted from the execution of the program is called *dynamic*. It is based on runtime information, such as content of variables, conditions executed, stack size and so on. This kind of data is difficult to obtain because of the lack of mechanisms to gather information from the program memory. Based on the work from Diehl [8], the next subsections explain how *dynamic* visualization is done.

2.3.2.1 How data is collected?

This is the most important aspect to consider when doing visualization of dynamic sources. In almost every case, the mechanism to obtain data must be invasive, i.e. some instructions must be placed in the original code in order to gather information. The intrusion in the source code will make the software harder to maintain as well as it will slow down the performance depending on the quality and what type of data is being extracted.

Recently there has been some new techniques not explored yet, this is the case of aspect-oriented programming (AOP). It is being used to trace, debug and profile programs. The powerful capabilities of AOP may be used to gather information about the execution of programs without being too invasive, avoiding the problem that a development statement gets executed in production environments.

2.3.2.2 What data is collected?

Typically, the data that users want to see in the visualization are the values of variables within a method during its execution. However, there are many other variables that would be possible to see if a mechanism to extract them is provided, such as program counters, number of line that is being executed, execution stack, memory allocated and so on. The visualization of each type of values depends on the existence of a method to gather it.

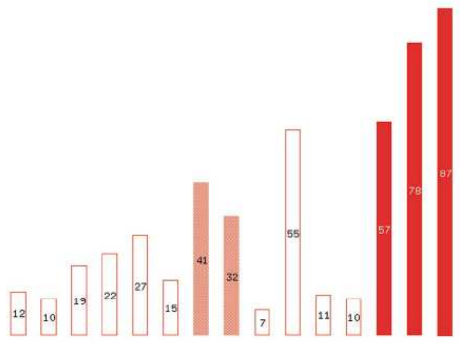


Figure 2.4: X-Tango Visualization

2.3.2.3 Fundamental Techniques

The intuitive technique to visualize runtime information is the animation. Although this method is easy to imagine, most of the time it does not fit in all kinds of dynamic information. One example to visualize dynamic information without using an animation is to calculate the average value of variable through time and print it. Although it will not be a really meaningful way to show the behavior a program, it could be enough in particular cases. Another option is to use a XY plane to plot the value of a variable through time, this is a better option but the 2D world offers more than that.

With two dimensional views is possible to create more elaborated views with geometrical shapes changing through time. Two good examples of the use of this strategy are X-Tango [36] (figure 2.4) and SAMBA [35] (figure 2.5). SAMBA proposes a language to visualize runtime information by generating a series of commands from the evaluated program (which means that is highly invasive) and passing them through a tool that draws the result. In this case the tool is called to be offline i.e. is not linked with the program under execution. If the tool visualizes the information gathered from the program while it is running, the tool is *online* with the program.

Proposals such as the one presented in [4] tries to involve 3D space in dynamic visualization. It tries to visualize the evolution of a sorting algorithm through time. In this case the values to be sorted are represented by 3D bars where the height represents the value. Once an iteration is complete, these bars are changed to a new position to indicate that a set of elements were sorted. The evolution of the algorithm is seen by placing each iteration result in the z axis of a 3D world. Thus keeping history of how the program got executed to solve the problem. This example can be seen in figure 2.6.

2.3.3 Evolutionary Information Source

The study of software evolution is a complete research field. Regarding the visualization of data resulting from the analysis of this source of information, several interesting proposals have been written. In [12] was proposed that the basic unit of information needed to visualize software evolution is called a maintenance request (MR). It refers to the delta of change in a software system e.g., a committed revision in a control versioning system like SVN or CVS. These deltas are taken together to analyze and visualize the evolutionary information about a software system.

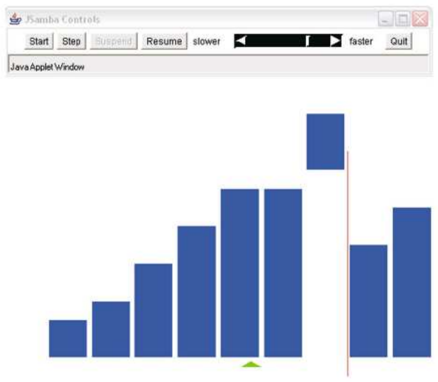


Figure 2.5: JSamba in action

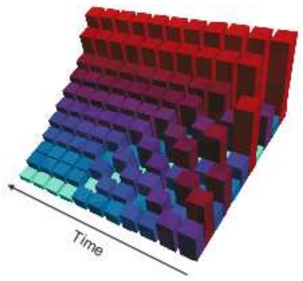


Figure 2.6: Bubble sorting using a three dimensional space

2.3.3.1 Software Metrics through Time

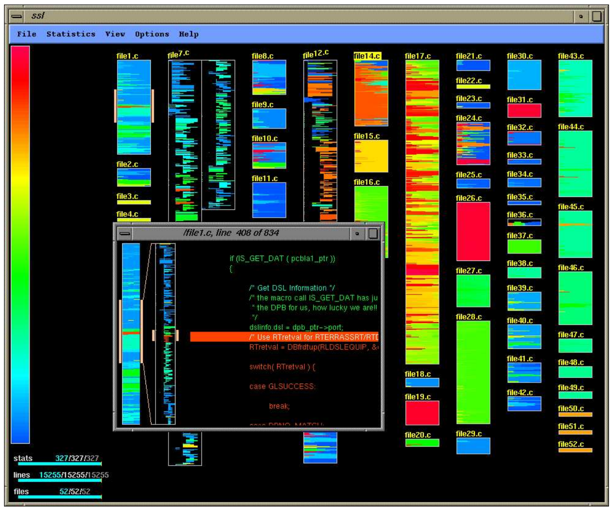
As explained in the previous sections, a software metric is a measure of some particular property of the software system. From the evolution's point of view, this metric may change its value through time. This change is what the visualization of evolutionary information looks for.

In this area, SeeSoft [9] is the most representative example and also probably the most successful visualization tool. It uses a metaphor based on rectangles to represent each file in the system, each row in the rectangle corresponds to a line of code and the color of the row represents how often the line has changed. Figure 2.7 shows a view of the system.

Particularly when visualizing metrics through time, the third dimension has not been widely explored; instead the efforts have focused in developing metaphors based on rectangles, shapes, sizes and colors. However Xie *et al.* on [43] proposed a visualization tool that uses the sv3D metaphor to visualize the evolution of a software system. The tool is called cv3D because of the relation with the control version system CVS. Due to the flexibility of the metaphor, it is possible to visualize the sequences of changes over the time.

2.3.3.2 Visualization of Structural Change

Many of the tools developed to visualize software have focused on the structure of the code by showing relations or dependencies on it. But few of them provide information about how these structures changed over time [7]. About this particular kind of visualization, there is a clear need of an additional dimension, that could be a spatial dimension (as it is shown in figure 2.8) or a temporal dimension, since the structure itself



```
if (IS_GET_DAT ( pcbla1_ptr ))
{

    /* Get DSL Information */
    /* the macro call IS_GET_DAT has ju
     * the DPB for us, how lucky we are!!
     */
    dslinfo.dsl = dpb_ptr->port;
    /* Use RTretval for RTERRASSRT/RTD
    RTretval = DBfrdtup(RLDSLEQUIP, &

    switch( RTretval ) {

    case GLSUCCESS:

        break;
```

Figure 2.7: SeeSoft Example

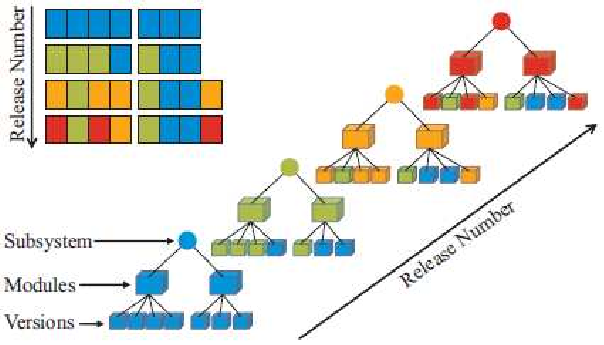


Figure 2.8: Structural change through time

uses at least two spatial dimensions. These structural views of the system can be seen as architectural views, so they have the same problems as the ones exposed previously (need to be built from small components).

2.3.3.3 Visualization of Evolutionary Coupling

One property that can only be extracted from the software evolution information is about determining how the files in the source code are related with each other. In this case, the files can be seen as entities that change at the same time with others, i.e. two changed files are committed to the repository at the same time. This property makes them coupled and hence the name of evolutionary coupling.

Zimmermann in [44] introduced the concept of Support Graph. This is a method to indicate relationships among software artifacts, as it can be seen in figure 2.9. In this visualization a pair of files from the Mozilla Firefox project is coupled if there is a line between them, and the proximity of them represents the weight of the relation. This way emerge a set of clusters representing how the source code is organized.

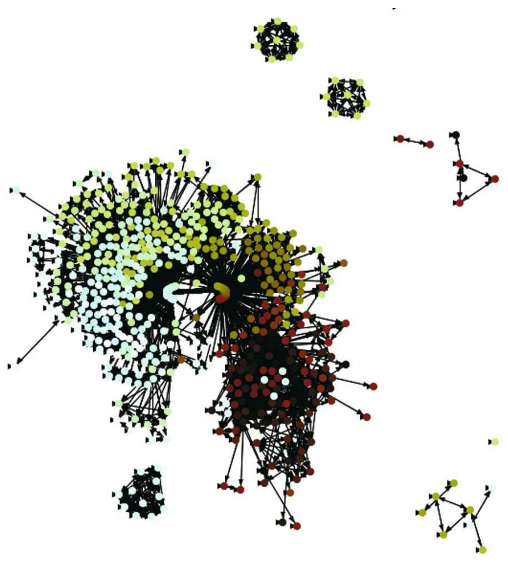


Figure 2.9: Support graph of Mozilla Firefox

2.4 UML and Software Visualization

UML [33] was first conceived as a union of the methodologies proposed of Grady Booch, Ivar Jacobson, and James Rumbaugh. They participated in the evolution of what was the first UML specification which came out in 1997 under the organization of the Object Management Group[2]. This language clearly follows the same direction of software visualization. The main difference is the way they face the problem, while software visualization tries to provide a view of an already written software system, the UML specification aims to provide a view before the actual code is written in order to avoid waste of time and resources. UML is a modeling language and also a visualization tool, although it does not provide a view of a system in terms of its metric values. Despite of the effort invested in the modeling stage of any methodology, there will be always the need to determine the current state of a system. This current state and its metrics can be determined by a visualization tool that analyzes the written source code.

2.5 What has been done?

Software visualization systems have been developed since it was necessary to understand software in an easier and better way. As can be seen through this chapter, many techniques and tools have been proposed to face the problem. Therefore, it could be helpful to see what has been done in the area, but this task would generate a huge table of tools developed until now.

The following table contains some important tools that have some level of popularity and acceptance in the community. The table 2.2 is a non-exhaustive list of the main tools developed in the field.

Table 2.2: Non-Exhaustive list of Applications

[2]After a negotiation for the “UML” name with Rational

Tool	Description	Visualization Techniques	Source of information
sv3D [28]	It uses three dimensional polycylinders as a metaphor	It uses 3D, Colors, Heights and deepness to show information.	Static
Vizz3D [23]	Visualization as Cities	It uses color, heights and a real metaphor to show software components and their relations.	Static
Tarantula [18]	It uses a SeeSoft like representation to show the results of a set of tests	It uses color and geometrical shapes to show the results	Dynamic
STAN[3]	Visualization of Java dependencies at different levels of abstractions	It uses a two dimensional metaphor, based on the Eclipse platform set of icons and graphs to show relations	Static
Structure 101[4]	Visualization of Java dependencies at different levels of abstractions	It uses a two dimensional metaphor, based on the Eclipse platform set of icons and graphs to show relations	Static
Code Crawler [21]	Visualization of code using polymetric view	It uses a two dimensional approach to show relations and metric	Static
SeeSoft [9]	Visualization of changes through time	It uses color and geometrical shapes to show the results of the analysis	Evolution
X-Ray [27]	It visualizes relationships in Java source code	It uses geometrical shapes, links (lines) and some basic colors to represent dependencies	Static
CVScan [40]	It shows information extracted from the CVS repository	It uses rectangles and color to represent information	Evolution
EPOSee [3]	It shows information about the evolution of files in the source code.	Uses pixel-map and support graph representations to show relations among files in the version control system	Static
SHriMP [38]	It shows the dependencies in a program code and other kinds of artifacts like architectural design and documentation	It uses a two dimensional approach to show the analyzed information. It is based on rectangles, color and links among the parts of the visualization	Static
X-Tango [36]	It visualizes the execution of a program	It uses an animated version of geometrical shapes and color to show the behavior of the program	Dynamic

[3]http://stan4j.com/
[4]http://www.headwaysoftware.com/products/structure101/index.php

2.6 Summary

This chapter presented the background in software visualization. First, the objective and fundamental challenges of software evolution were explained, so it was possible to identify the need for software understanding techniques, such as software visualization. Then, the concepts on software visualization and how they are applied to the main sources of information were explained. It was also clarified how software visualization differs from other techniques like UML. Finally, a non-exhaustive list of software visualization tools was presented to indicate the state of art in the field.

Chapter 3

SeeIT 3D

SeeIT 3D stands for *S*oftwar*e* visualization *E*clipse *I*ntegrated *T*ool 3D. It is a tool that allows the user to perform three dimensional software visualization within an Integrated Development Environment (IDE). This tool pretends to facilitate the understanding of a software system by visualizing design errors or bad smells in code. Its objective is to provide a mechanism that allows the user to understand how a software system was built.

SeeIT 3D is developed taking the founded bases in the software visualization area, so it avoids starting the development of a new approach from scratch. Therefore, SeeIT 3D is able to take the field a step further by providing a set of characteristics based on previously developed approaches. Considering this previous work, the metaphor proposed by Marcus *et al.* in [28] was taken as a base since it provides a flexible mechanism to show a considerable amount of data, which can be extracted from different sources of information as it was presented in [43].

A second objective of SeeIT 3D, is to be focused on facing the challenges proposed by software evolution as it was presented in chapter 2. The first challenge refers to the support for multiple languages: choosing and IDE capable of supporting many languages and making the metaphor independent from the source of information this challenge can be met. The second challenge is related to the manipulation of huge amounts of information presented to the user: a three dimensional approach is able to show more information than the vast majority of visualization tools that use a two dimensional approach. Finally a third challenge is addressed by the tool: teaching software evolution with the help of a visualization tool makes easier to understand the concepts behind a software system.

The construction basis, the architecture, the metaphor as well as the technologies used are presented in the following sections. Additional information about the development of SeeIT 3D can be found in appendix A.

3.1 The Metaphor

SeeIT 3D is based on the metaphor proposed by Marcus *et al.* in [28]. This metaphor is able to handle high amounts of data from different sources of information and it offers high flexibility by using a third spatial dimension. This allows the user to navigate, explore and change the mapping between software artifacts and visual properties that include the color, height and width of the objects. The metaphor is based on the concept of *polycylinders*: three dimensional bars with polygonal base. When they are grouped they represent a set of related artifacts that build a *container* where graphical properties are used to represent the software system.

Formally SeeIT 3D defines its metaphor as a triple $P = \{A, V, M\}$ where:

- $A = \{a_1, a_2, ..., a_n\}$ is the set of artifacts to analyze
- $V = \{v_1, v_2, ..., v_s\}$ defines the elements of the visual metaphor used. Each v_j is composed by:
  - *Polycylinders* - p
  - *Containers* - c
  - *Polycylinder height* - h
  - *Polycylinder width* - z
  - *Polycylinder color*- o
  - *Container relationships representation*- r
- Each $a_i \in A$ is represented by a *container* c_i, composed by *metrics* $t_i = \{t_{i1}, t_{i2}, .., t_{im}\}$ and *polycylinders*$p_k \in c_i$.
- Each *polycylinder* $p_k \in c_i$ represents a finer granularity level of the artifact represented by the *container* c_i. It also has information about metric values v_{km} (where k refers to the *polycylinder* and m to the associated metric).
- M defines the mapping between data and visualization as a set of relations $m_p \in t_i \times V$

According to the visual elements defined by V, every visualization instance is formally specified as:

$$visualization = \{c, p, h, z, o, r\} \quad and \quad \exists m_p \in M \tag{3.1}$$

The above definition and its elements are illustrated in figure 3.1.

It is important to clarify that SeeIT 3D has a fixed number of visual properties, i.e. it is not variable like the number of attributes in the analyzed data. This forces the user to decide which attribute is going to be represented when the number of attributes is higher than the number of visual properties. The same occurs when the number of visual properties is higher than the number of attributes.

Table 3.1 shows a set of possible visualizations as it is defined in equation 3.1. It specifies all the visualization elements based on artifacts of a software system written in Java.

As it was explained previously, SeeIT 3D employs a powerful and flexible metaphor that enables it to perform a wide variety of visualization types, with information gathered from different sources of information. This characteristic makes the visualization system a useful tool for the software developer who seeks for a way to improve his code, correct bugs and acquire a wider knowledge of the system.

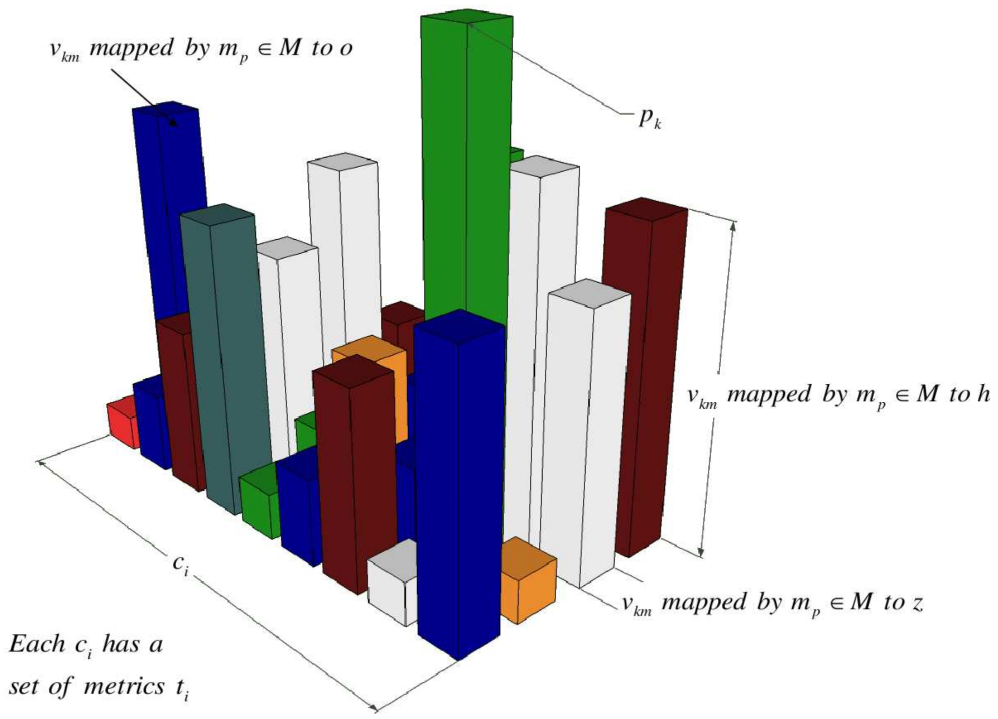

v_{km} mapped by $m_p \in M$ to o
p_k
v_{km} mapped by $m_p \in M$ to h
v_{km} mapped by $m_p \in M$ to z
c_i
Each c_i has a set of metrics t_i

Figure 3.1: SeeIT 3D metaphor elements

Table 3.1: Metaphor use. Concrete example

Container	*Polycylinders*	*Height*	*Width*	*Color*	*Visual relationships*
Package	Classes	LOC	LCOM	Complexity	Lines
Package	Classes	Complexity	LCOM	LOC	*Container* mark
Classes	Methods	LCOM	Complexity	LOC	Lines
Package	Methods	LOC	LCOM	Complexity	No relation
Package	Lines	LOC	-	Control structure	Clustering
Project	Classes	Complexity	LCOM	-	*Container* mark

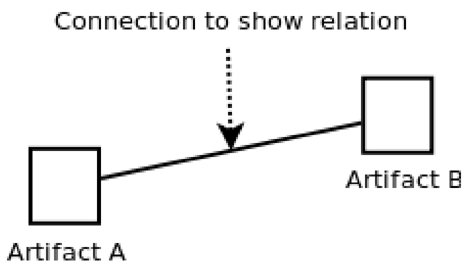


Figure 3.2: Bidimensional relationship example

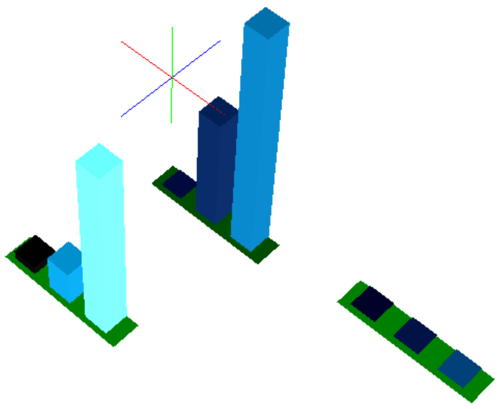


Figure 3.3: Visualizing relationships by using *Common Base*

3.1.1 Relationships among Containers

The lack of a mechanism to display relationships among containers in the sv3D metaphor is faced by SeeIT 3D. Many ways to show relationships among artifacts in a bidimensional world has been developed.

Graph-like representations, as the one illustrated in 3.2, have been mostly used in these cases. These mechanisms are based on geometrical shapes and lines, proximity or hierarchical approximations to display relationships between two or more artifacts. They have been suitable to show information due to their simplicity and understandability. However, when visualizing huge amounts of information there are advantages of one mechanism over the other. For example, the use of proximity among related artifacts allows us to understand relationships easier than when using lines to connect geometrical shapes.

Taking into account the fact that there is no single method powerful and flexible enough to handle different amounts of information and types of relationships, SeeIT 3D provides a set of mechanisms to display them. It has been defined four visualization metaphors to this end: Common base, Lines, Arcs and Motion based. They are described next:

- *Common base* displays a base under each related container. The root container, i.e. the one that has relation to the others is rendered with a darker color to indicate that it is the source of relationships. Figure 3.3 illustrates this concept using a container that has two related containers. By using this mechanism is possible to visualize great amounts of related artifacts without confusing the user.

- *Lines* are the basic mechanism used when viewing relationships among elements. As it is shown

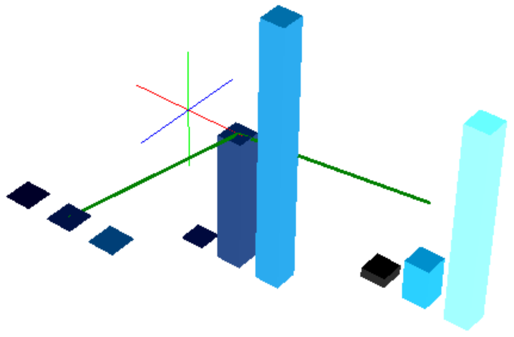


Figure 3.4: Visualizing relationships by using *Lines*

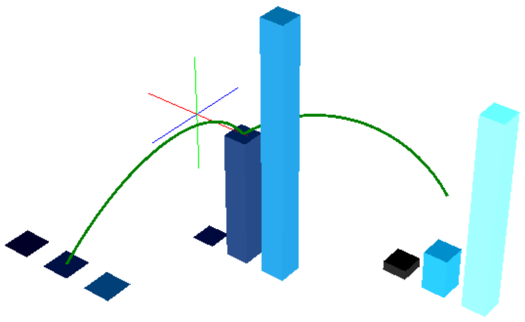


Figure 3.5: Visualizing relationships by using *Arcs*

in figure 3.4, by using this kind of representation is possible to easily see the relationships among containers but it fails when there are too many related containers.

- *Arcs* provide a way that is easier to understand than previous representation. As it is depicted in figure 3.5, arcs are easier to follow than lines because they are less intertwined. This is a more general way to indicate relationships among containers.

- *Motion based* has not been used before as a way to represent relationships. It is based on movement to highlight the related objects. This movement is easily perceived by the user since the human eye is sensitive to it. By using this mechanism, it is possible to visualize huge amounts of information without confusing the user as in the case of lines or arcs. Additionally it is less invasive in the metaphor because it does not add extra elements to it. Figure 3.6 illustrates this concept.

3.2 Underlying Technologies

In software development the success of a new product is conditioned by the correct use of the technologies that build it. In the case of SeeIT 3D, these technologies need to be chosen carefully in order to allow the building process to be completed with high quality. The technological base for SeeIT 3D is presented in the following sections.

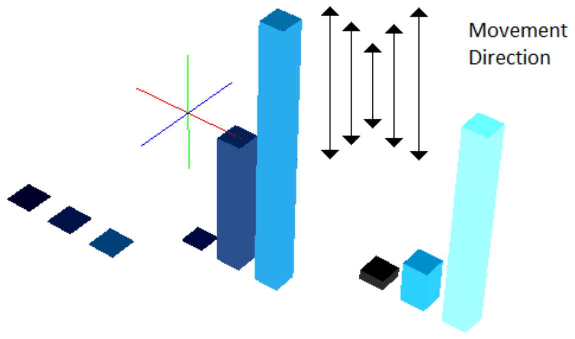


Figure 3.6: Visualizing relationships based on movement

3.2.1 The IDE

Nowadays in developers' community, there is no tool more necessary to work than an IDE, since it provides a lot of utilities and allows developers to write code in a faster, safer and cleaner way.

There has been developed several IDEs in recent years. Visual Studio[1] suite and MonoDevelop[2] are examples for writing software in C# programming language with target to the .NET framework. In the case of the Java programming language, there has been developed many other IDEs, such as JDeveloper[3], JBuilder[4], NetBeans IDE[5] and Eclipse[6]. Besides the ones mentioned, there are IDEs for almost every programming language, all of them serving for the same purpose: to provide an editor, compiler and a debugger.

However, there are some differences among them that make one more accepted than the others. Some of the key factors when choosing an IDE are: functionality available, support, cost, learning curve, and recently, one key aspect is the support for extending its functionality. The latter aspect, as well as the support for a popular programming language like Java, has resulted in an advantage for Eclipse IDE over the others. This fact can be observed in figure 3.7 that is based on the result of searching each IDE at Google Trends. In this figure, Eclipse (in Blue) is the most popular IDE chosen by developers. This tendency and the support expressed by Mens *et al.* in [30], make Eclipse the chosen one to work with.

3.2.2 The Graphics Engine

Since SeeIT 3D is based on a three dimensional metaphor, it needs to be built on top of a graphics engine that takes care of the rendering process. Considering that the language exposed by the Eclipse platform to extend its functionality is Java, in the following subsections the three main possibilities to write applications using a third spatial dimension are presented.

[1]http://www.microsoft.com/visualstudio/
[2]http://monodevelop.com/
[3]http://www.oracle.com/technology/products/jdev
[4]http://www.embarcadero.com/products/jbuilder
[5]www.netbeans.org
[6]www.eclipse.org

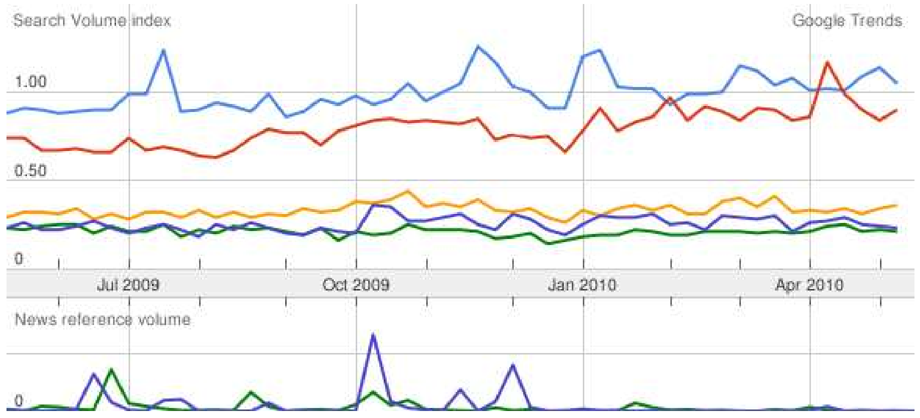


Figure 3.7: Google trends results about downloads of the most popular IDEs. Blue: Eclipse, Red: Visual Studio, Orange: NetBeans, Green: JDeveloper, Purple: JBuilder

3.2.2.1 JOGL

JOGL is one of the open source technologies initiated by the Game Technology Group at Sun Microsystems. JOGL provides full access to the APIs in the OpenGL 2.0 specification, as well as vendor extensions, and can be combined with AWT and Swing components. JOGL has the same focus as OpenGL on 2D and 3D rendering. Most features of the popular OpenGL GLU (the OpenGL Utility Library) and GLUT (OpenGL Utility Toolkit) libraries are present in JOGL. GLU includes support for rendering spheres, cylinders, disks, camera positioning, tessellation, and texture mipmaps. The JOGL version of GLUT does not include its windowing functionality, which is handled by Java, but it does offer geometric primitives (both in solid and wireframe mode). JOGL's utility classes include frame-based animation, texture loading, file IO, and screenshot capabilities. JOGL has evolved into the reference implementation for the JSR-231 specification for binding OpenGL to Java (`http://jcp.org/en/jsr/detail?id=231`). JOGL 1.1.1 was superseded by JSR-231 in October 2005, and the JSR-231 is the current release which came out in May 2008 [6].

The OpenGL API is accessed via Java Native Interface (JNI) calls, leading to a very direct mapping between the API's C functions and JOGL's Java methods. The drawback is that the OpenGL programming style is based on affecting a global graphics state, which makes difficult to structure Java code into meaningful classes and objects. JOGL does provide class structuring for the OpenGL API, but the vast majority of its methods are in the very large GL and GLU classes. OpenGL is a vast, complex, and powerful API, with entire books dedicated to its explanation [6].

3.2.2.2 Scene Graph Based Engines

A scene graph makes 3D programming much easier because it emphasizes scene design, rather than rendering, by hiding the graphics pipeline. The scene graph is a structure that stores data about the world. The relationships between scene data (geometric, sound, physical, etc.) are kept in a tree structure, with leaf nodes representing the core elements. These core elements typically are the ones rendered to the scene. Organization of the scene graph is very important and it generally depends on the application.

The scene graph has two main advantages: it simplifies 3D programming and it speeds up the resulting code. The scene graph hides low-level 3D graphics elements and allows the programmer to manage and organize a 3D scene.

Java 3D

The Java 3D API, a scene graph API developed by Sun Microsystems, provides a collection of high-level constructs for creating, rendering, and manipulating a 3D scene graph.

At the Java 3D implementation level, the scene graph is used to group shapes with common properties and carry out view culling, occlusion culling, level of detail selection, execution culling, and behavior pruning. Java 3D utilizes Java's multithreading to carry out parallel graph traversal and rendering, both very useful optimizations [6].

Java 3D is designed with performance in mind, which it achieves at the high level by scene graph optimizations, and at the low level by being built on top of OpenGL or DirectX Graphics. Some programmer-specified scene graph optimizations are available through capability bits, which state what operations can/cannot be carried out at runtime (e.g., prohibiting a shape from moving). Java 3D also permits the programmer to bypass the scene graph, which gives the programmer greater control over rendering and scene management [6].

Finally Java 3D is one of the engines with more documentation available on the Internet, most books about it and a very well-formed forum with people always available to help.

jMonkey

jMonkey Engine (jME) is a high-speed real-time graphics engine. It is mainly used to develop games because it provides an infrastructure to do so. During the last year it has been gaining attention because of its well written API and good support on its community forums. All the characteristics that applies to Java 3D are also applicable to this engine.

3.2.2.3 Choosing the Graphics Engine

The development of a visualization tool like SeeIT 3D is focused on helping developers to understand the code they are writing, rather than making a research on computer graphics techniques. With this in mind and considering the complexity of JOGL, it is discarded because it applies in a lower level that is out of scope in this project.

This leaves the selection between Java3D and jME. The jMonkey Engine site states about it is aimed to game development, it causes not to fulfill the needs of SeeIT 3D because it is intended to work under an IDE and not to work with the concept of a game. Also jME does not have all the documentation and support that Java3D does with Sun behind its development. As a consequence, the graphics engine of choice is Java3D that fits with the concept behind SeeIT 3D and takes the complexity of rendering an image apart, leaving the focus on the core functionality of the tool.

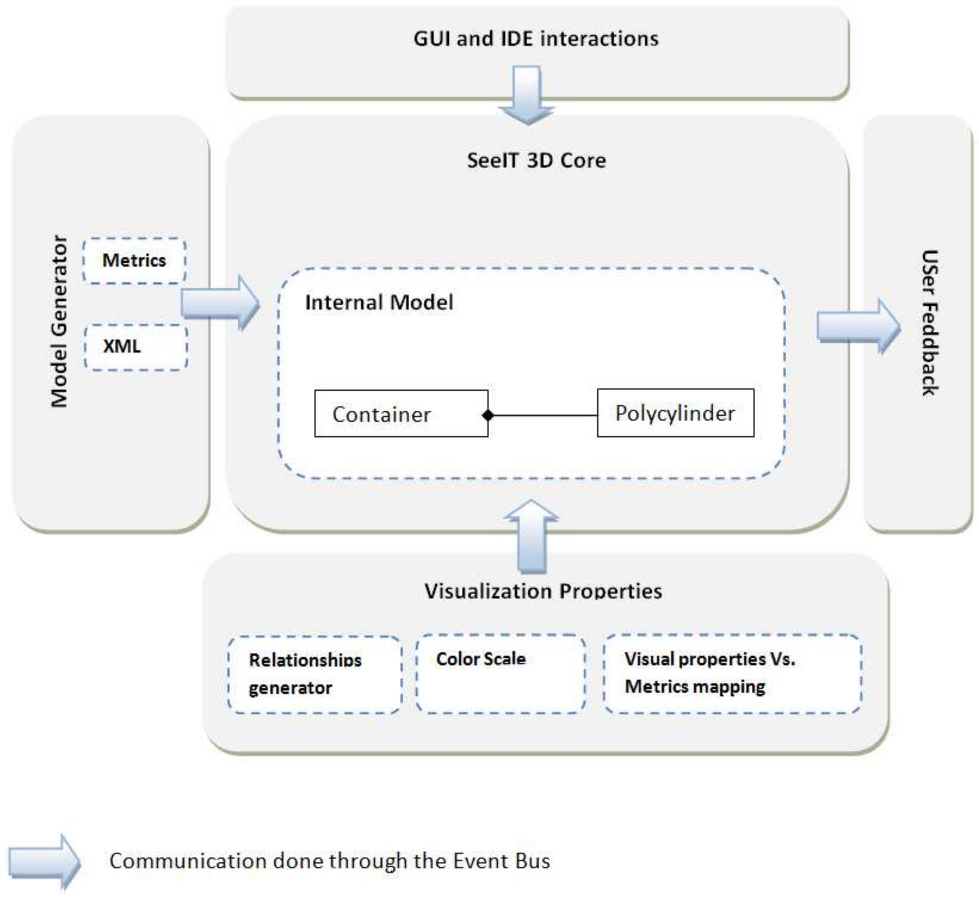


Figure 3.8: SeeIT 3D internal architecture

3.3 Internal Construction

SeeIT 3D is built on top of the Eclipse platform for building plugins. This platform gives several options and controls to extend the functionality of the IDE.

The visualization process starts with the source code analysis. It is based on the model exposed by the JDT (Java development tools)[7] plugin, which centralizes the information about the source code within the IDE. Once the analysis is done, the SeeIT 3D *core* is responsible for building the *scene graph* that will be taken by Java 3D. The scene graph generation process is based on the construction of small parts, which together built the whole graph. This way, each *polycylinder* generates its own part, which is then passed to the parent *container* where a bigger part of the graph is built upon the segments provided by all the *polycylinders*. The assembled parts are sent to the *core*, which is in charge of composing the complete *scene graph* and add the enabled user interactions. Later in the visualization process, these interactions will trigger events that are published in the *Event Bus* where the right handler takes care of processing them.

The components of SeeIT 3D can be observed in figure 3.8. This figure illustrates the five main components: *GUI and IDE interactions, model generator, user feedback, visualization properties* and *SeeIT 3D core*. These components are described next:

[7]http://www.eclipse.org/jdt/

- *GUI and IDE interactions:* This module is in charge of handling all the interactions performed through the exposed user interface within the IDE. These include the visualization commands, shortcuts and icons. The interactions performed directly over rendered image are handled by a special component in the scene graph, which is in charge of redirecting them to the right handler.
- *Model generator:* This module generates information that is taken by the *core* to build the visualization. The generation process is built from a generic mechanism that allows analyzing information as long as it conforms to a defined specification exposed by the module. Also within this module the metrics calculation is performed.
- *User feedback:* This module is a set of interfaces that enable the core to send relevant information to the end user, in order to keep him up to date with the visualization state.
- *Visualization properties:* It defines how the visualization is going to be generated. It includes the color scale, current mapping and visual relationships generator.
- *SeeIT 3D Core:* The core is in charge of keeping track of the visualization state. Within this module lies the internal model of the tool, where all the incoming information from other modules is stored and then reflected on the screen.

3.3.1 Metrics

Visualization with SeeIT 3D is based on the metrics values of a software artifact. As it was mentioned in chapter 2, a software metric indicates the level of a property the software system does or does not have.

In SeeIT 3D the metrics are separated into two categories based on their values, *numeric* or *nominal*. The *numeric* category refers to metrics whose values are numbers in a predefined range e.g. from 0 to 10. On the other hand *nominal* metrics are the ones whose values are not numerical, instead they are based on a predefined set of categories e.g. Good, Bad. Table 3.2 describes the metrics used by SeeIT 3D.

A second way to classify metrics is also shown in table 3.2. This classification is based on the type of analysis they perform in the analyzed artifact: *Syntactic* and *Semantic*. *Syntactic* analysis takes into consideration the structure of the artifact itself i.e. how the artifact is built and under which rules. On the other hand, *Semantic* analysis considers the content and quality of the artifact. For example, the number of lines of code is considered syntactical analysis because it only depends on the structure of the artifact, while the lack of cohesion metric analyzes how the artifact accomplish its objective.

The metrics maximum values were determined according to the definition of the metric itself or based on the experience of the author. Specifically, the McCabe complexity [29] states that "an upper limit of 10 for program complexity is proposed because greater complexity would be less manageable and testable"; this limit was adopted by the National Institute of Standards and Technology (NIST[8]) of the United States which confirms its usefulness. Within SeeIT 3D was adopted the value of 11 to indicate that values beyond 10 should be considered as harmful for the code.

The calculation of Lack of Cohesion Metric (LCOM) is performed as it is depicted in equation 3.2, where n is the number of methods, a the number of attributes and $m(a_i)$ is the number of methods that use the

[8]http://www.nist.gov/

Metric name	Description	Type (based on value)	Type (based on type of analysis)	Maximum value
LOC	Indicates the number of lines within a software artifact (method, class, package, project)	Numeric	Syntactic	400
McCabe complexity	It is based on the McCabe complexity [29] for methods. At class level the metric calculates the average value of its methods. At package level the metric calculates the average value of its classes.	Numeric	Semantic	11
LCOM	Calculates the Lack of Cohesion metric proposed in [13] for classes. Not functional at method or package level.	Numeric	Semantic	2
Control Structure	Indicates the control structure present in a line of code. Its value varies in *for*, *while*, *if*, *else* and *none* when there is not a control structure.	Nominal	Syntactic	Does not apply maximum but number of categories

Table 3.2: Metrics used in SeeIT 3D

a_i attribute of the class. A class with high level of lack of cohesion is that which its methods use a few or none attributes defined by the class. This way, the equation 3.2 is reduced to evaluating the fraction $\frac{n}{n-1}$, such equation is always evaluated in the range of (1, 2] for $n > 2$ and 0 for other values.

$$LCOM = \frac{n - \frac{\sum_{i=1}^{a} m(a_i)}{a}}{n-1} \tag{3.2}$$

Finally, the number of lines of code was limited to 400 based on the experience of the author; an artifact with more than 400 lines needs to be highlighted by the tool so the user can know about it.

These metrics calculation is performed with the support of the JDT plugin. This plugin provides a convenient model that can be used to extract information easily from the source code written in Java. The metric value is then processed by a normalizer in order to get a value in the range [0, 1]. Once the value is normalized, it is possible to bind it to a visual property like color, height or width. Using this mechanism, the consistency of the visualization is maintained by representing metrics whose values range in different intervals.

3.4 XML Based Visualization

With SeeIT 3D is not only possible to visualize the source code within the Eclipse IDE. The tool provides a XML specification that allows other analysis tools to build a XML file containing the necessary information to render it on the screen using the metaphor.

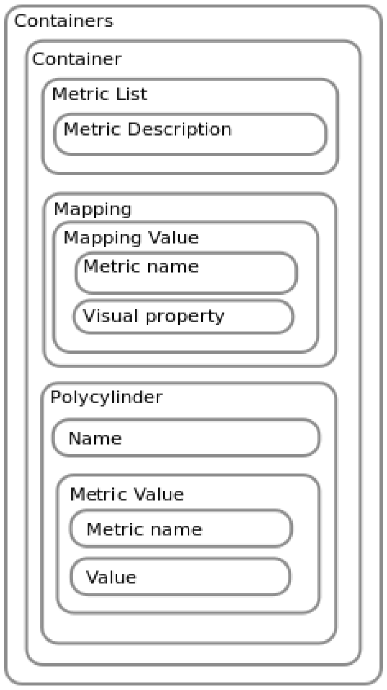


Figure 3.9: Information structure of a XML based visualization file

The XML specification is flexible enough to allow different kinds of information like static, dynamic or evolutionary to be visualized within the IDE, thus eliminating the limitation of other visualization tools that only allow the analysis and visualization of certain type of software artifacts.

The XML format was chosen because of its maturity in the software industry as a way to exchange information between systems, besides providing a human readable syntax that allows comprehending the structure of the information as well as simple error detection.

In figure 3.9 is shown how the information is structured in order to be read by SeeIT 3D and then visualized with the metaphor of the plugin. This structure reflects the definition of the metaphor, as it was explained in section 3.1.The XML-Schema for the XML file can be found at `http://seeit3d.googlecode.com/files/seeit3d.xsd`. This type of visualization allows the same interaction level as in the case of source code visualization[9]. Appendix B lists an example of a file with this structure.

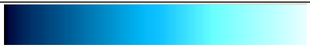
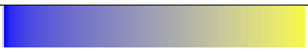
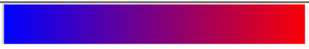
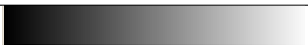
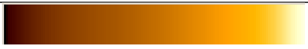
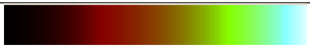
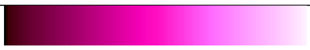
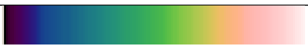
3.5 Color Scales

Color scales have been widely used to represent numerical ranges in different research and production fields. These scales are important to represent information that needs to be rapidly processed by the user and where a number is not representative enough under certain circumstances. For example when taking a quick look at a table results with lots of numbers.

These color scales should have at least three properties as stated in [22]:

[9]Except the Open Editor command and Highlight artifact in the *Package Explorer*

Table 3.3: Color scales used in SeeIT 3D

Name	Color scale (min to max, left to right)
Blue Tone	
Blue to Yellow	
Cold to Hot	
Gray Scale	
Heated Object	
Linear Optimal	
Magenta Tone	
Rainbow	

1. *Order* should be preserved between the numerical range and the color scale. This should be achieved by using a color which can be understood in a lower value than other,

2. *Uniformity and representative distance* means that colors should convey the distances between the values they represent i.e. colors representing values equally different from each other along the scale should be perceived as equally different in the color scale, and

3. *Boundaries* where the color scale should not create perceived boundaries that do not exist in the numerical data. That is, it should be able to continuously represent continuous scales.

SeeIT 3D has multiple color scales. They were chosen because of their fulfillment with the three properties or because of their common usage in other areas of knowledge. According to this, in table 3.3 are listed the color scales that can be selected by a user of the plugin.

3.6 Summary

This chapter presented the underlying concepts of SeeIT 3D, including the metaphor it is based on. The technologies used and why there were chosen was also presented. Then the principles of the internal architecture of the plugin and how the responsibilities are delegated to the corresponding component were explained. After, the metrics that allow visualizing information within the IDE were founded. In order to provide an extensible and external mechanism to visualize information, it was explained how the XML based visualization is performed. Finally the color scales were explained, and how their main characteristics must be established.

Chapter 4

Usability Concerns on SeeIT 3D

When developing a visualization tool, usability should be placed on top of its design and implementation. In SeeIT 3D the usability guided the work, even more when using a three-dimensional metaphor where objects need to be rotated, moved, zoomed and panned as user wants.

To complement the fundamental operations performed on a three dimensional view, the metaphor's flexibility must be supported by a well defined and easy to use GUI. To support this kind of operations, SeeIT 3D provides a set of user-customizable properties that can be changed to meet user's needs.

SeeIT 3D provides a wide variety of functionality that the user can use. It can be classified into two categories: *IDE interactions* and *Visualization interactions*. The former refers to those interactions the user is able to perform by using the controls provided by the IDE itself, while the latter contains those interactions the user is able to perform directly within the visualization. This chapter explains both of these types of interactions.

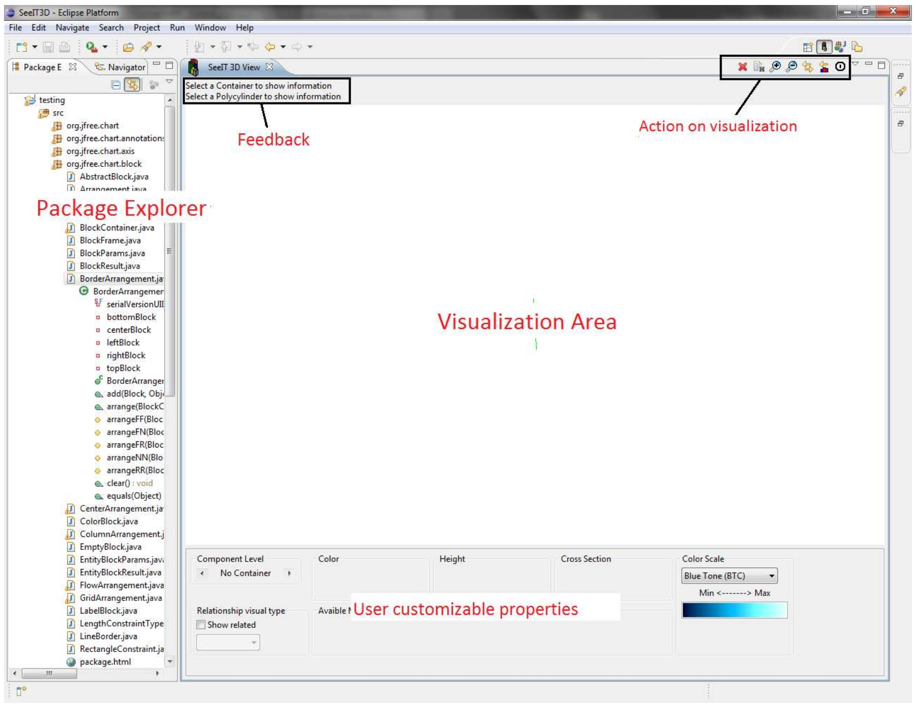
In figure 4.1 is shown the main screenshot of the tool. It contains the basic set of views assembled in a perspective that allows to the user maximum interaction with the different sources of information within the IDE.

4.1 IDE Interactions

Eclipse offers several standard controls and a set of methods to access information about source code and artifacts contained within the IDE. Taking advantage of these controls, SeeIT 3D functionality can be easily accessed from different places inside the IDE. It allows high level of interaction with the IDE and a customizable experience by allowing the user to change certain properties related to the visualization. They include:

- A custom *perspective* with a predefined layout with the necessary information that a user will need when using the tool. This perspective can be changed, saved and restored as user wants.
- Visualization of Projects, Packages, Java Files, Methods and XML files[1] present in the *Package Explorer view, Search Results view* and directly from the *Java Editor* as it is illustrated in figure 4.2.

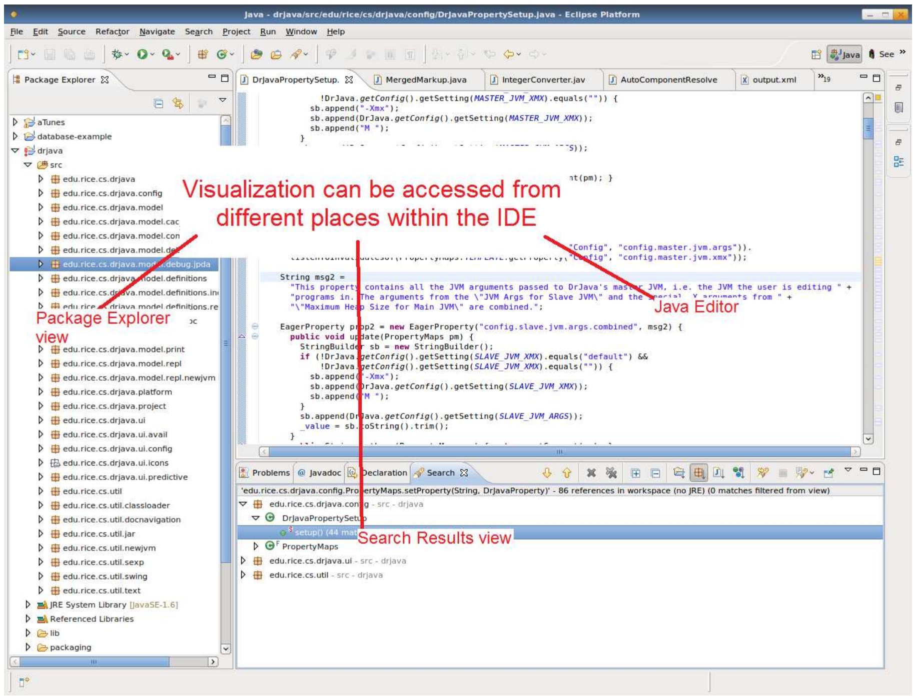
[1]with the appropriated structure



Select a Container to show information
Select a Polycylinder to show information

Component Level: No Container
Color Scale: Blue Tone (BTC)
Min <-------> Max

Figure 4.1: SeeIT 3D general view


Java - drjava/src/edu/rice/cs/drjava/config/DrJavaPropertySetup.java - Eclipse Platform
Visualization can be accessed from different places within the IDE
Package Explorer view
Java Editor
Search Results view
Problems
Javadoc
Declaration
Search

```
        !DrJava.getConfig().getSetting(MASTER_JVM_XMX).equals("")) {
      sb.append("-Xmx");
      sb.append(DrJava.getConfig().getSetting(MASTER_JVM_XMX));
      sb.append("M ");
    }

  String msg2 =
    "This property contains all the JVM arguments passed to DrJava's master JVM, i.e. the JVM the user is editing " +
    "programs in. The arguments from the \"JVM Args for Slave JVM\" and the special -X arguments from " +
    "\"Maximum Heap Size for Main JVM\" are combined.";

  EagerProperty prop2 = new EagerProperty("config.slave.jvm.args.combined", msg2) {
    public void update(PropertyMaps pm) {
      StringBuilder sb = new StringBuilder();
      if (!DrJava.getConfig().getSetting(SLAVE_JVM_XMX).equals("default") &&
          !DrJava.getConfig().getSetting(SLAVE_JVM_XMX).equals("")) {
        sb.append("-Xmx");
        sb.append(DrJava.getConfig().getSetting(SLAVE_JVM_XMX));
        sb.append("M ");
      }
      sb.append(DrJava.getConfig().getSetting(SLAVE_JVM_ARGS));
      _value = sb.toString().trim();
    }
```

'edu.rice.cs.drjava.config.PropertyMaps.setProperty(String, DrJavaProperty)' - 86 references in workspace (no JRE) (0 matches filtered from view)

Figure 4.2: Visualization availability within the IDE

- Customizable visualization properties for allowing changing the appearance at user convenience. These properties include:
  - Number of containers per row (when visualizing multiple containers)
  - Number of polycylinders per row on each container
  - Visualization background color
  - Highlight color of the visualization (current selection color)
  - Color of relationships (e.g. when using lines/arcs)
  - Scale up/down proportion (i.e. the percentage that the container will increase/decrease its size proportion)
  - Transparency proportion, as in the case of scale up/down proportion
  - Default color scale when the visualization is started
- Key binding for most of the commands. They allow navigating or changing the visualization by using a well known method for developers using the Eclipse IDE.

This tight coupling between the IDE and the tool allows the user to avoid propagating the changes into two places, the IDE and the visualization tool. This helps the tool to succeed where other visualization tools might have failed, reducing the associated cost of performing an additional task when developing software.

4.2 Visualization Interactions

This kind of interactions are the most important of SeeIT 3D since they allow the user to understand and find errors in the code. With the intend to provide enough tools for the user to visually analyze the information presented to him, SeeIT 3D provides the following set of possibilities to be used directly within the visualization:

- Selecting one or multiple containers and perform operations:
  - Deleting one or all containers in the view.
  - Scaling up or down the containers. An example can be seen in figure 4.3 where the same container is visualized with different sizes.
  - Linking container and polycylinder selection with package explorer. By linking the two views is possible for the user to detect which polycylinder represents which artifact, allowing to easily indicate what element posses some kind of problem or needs to be checked. This concept is illustrated in figure 4.4.
  - As a complement to the previous point, the user is able to open directly the related artifact by double clicking on the selected polycylinder, making the navigation between the view and the source code even easier.
  - Making more or less transparent a set of polycylinders for seeing hidden polycylinders.
  - Sorting the polycylinders within each container by height or color.

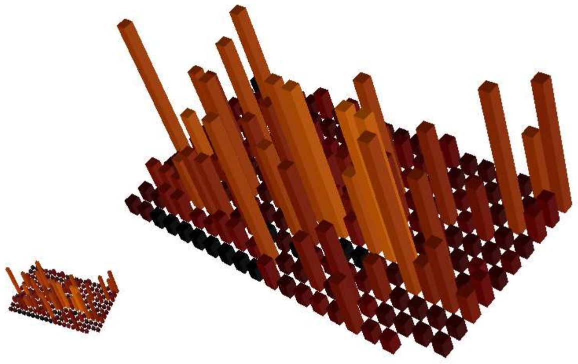
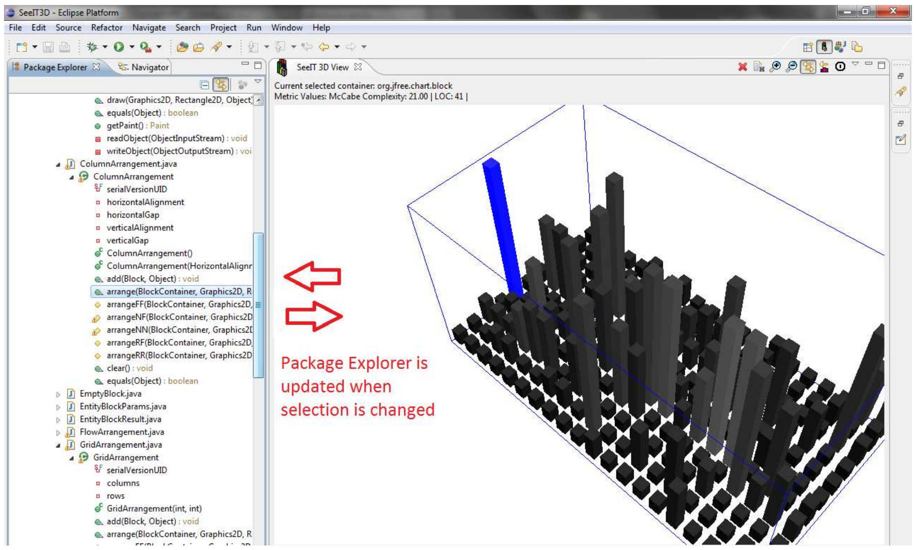


Figure 4.3: Scaling containers



SeeIT3D - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Navigator
SeeIT 3D View
Package Explorer is updated when selection is changed

```
draw(Graphics2D, Rectangle2D, Object)
equals(Object) : boolean
getPaint() : Paint
readObject(ObjectInputStream) : void
writeObject(ObjectOutputStream) : voi
ColumnArrangement.java
  ColumnArrangement
    serialVersionUID
    horizontalAlignment
    horizontalGap
    verticalAlignment
    verticalGap
    ColumnArrangement()
    ColumnArrangement(HorizontalAlignr
    add(Block, Object) : void
    arrange(BlockContainer, Graphics2D, R
    arrangeFF(BlockContainer, Graphics2D,
    arrangeNF(BlockContainer, Graphics2D
    arrangeNN(BlockContainer, Graphics2D
    arrangeRF(BlockContainer, Graphics2D
    arrangeRR(BlockContainer, Graphics2D
    clear() : void
    equals(Object) : boolean
EmptyBlock.java
EntityBlockParams.java
EntityBlockResult.java
FlowArrangement.java
GridArrangement.java
  GridArrangement
    serialVersionUID
    columns
    rows
    GridArrangement(int, int)
    add(Block, Object) : void
    arrange(BlockContainer, Graphics2D, R
```

```
Current selected container: org.jfree.chart.block
Metric Values: McCabe Complexity: 21.00 | LOC: 41 |
```

Figure 4.4: Package explorer vs. Visualization Synchronization

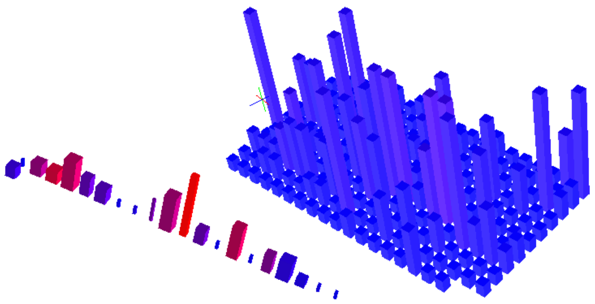


Figure 4.5: Different granularity levels

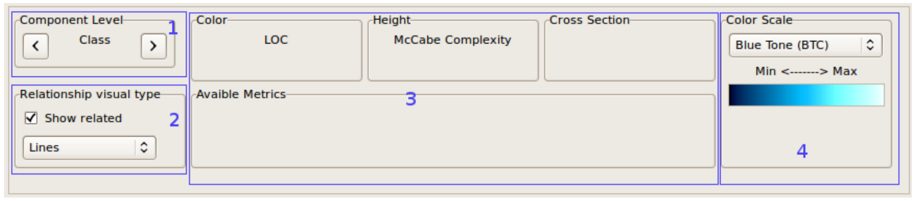
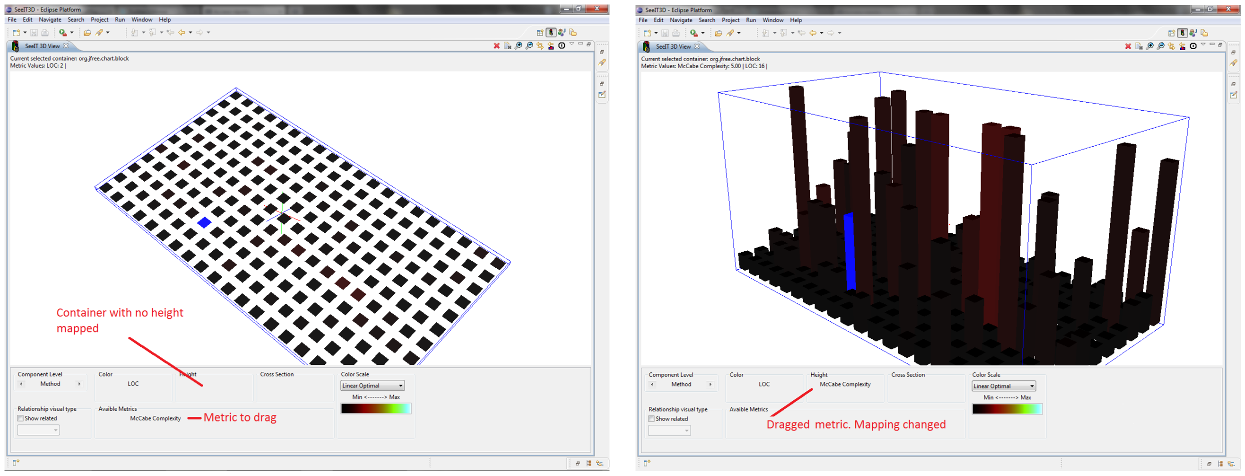


Figure 4.6: SeeIT 3D main interaction view


Container with no height mapped
Metric to drag
McCabe Complexity
Dragged metric. Mapping changed

(a) Before drag and drop action

(b) After drag and drop, mapping updated

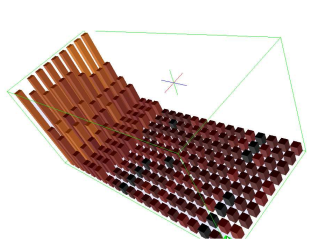
Figure 4.7: Drag and drop based mapping

- Changing granularity level. Figure 4.5 depicts the same container viewed at several granularity levels.
- Changing metric vs. visual property mapping.

- Allowing to save/load visualization state, avoiding redoing the analysis of one or several artifacts.
- Changing the color scale used in the visualization.

The previously stated interactions are performed by using the GUI. It was designed to be standard (including commonly used controls in other visualization tools) and easy to use and understand. In figure 4.6 is presented the main interaction view of SeeIT 3D, there can be changed the visualization properties. This view is splitted into four sections: the first one contains the possible granularity levels the user can select according to the selected container; the second section allows the user to select the mechanism used to render relationships; the third section allows to change the mapping between the visual properties (width, height and color) and the available metrics, by using the *drag and drop* concept (shown in figure 4.7). This method to update the mapping, allows the user to easily know what visual property is mapped to a certain metric besides making the process of changing the mapping simple and easy to understand; and the fourth section allows changing the color scale used in the visualization.

As mentioned in the characteristics list, SeeIT 3D provides mechanisms to allow filtering or selecting polycylinders in order to easily detect anomalies. These methods are*: polycylinder sorting* (figure 4.8a) and *polycylinder transparency* (figure 4.8b*).*


(a) *Polycylinder* sorting within a *container*

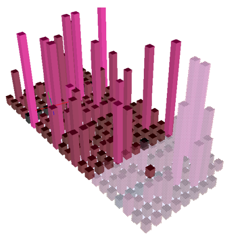

(b) Visualizing a container interior by using transparencies.

Figure 4.8: Information selection options

4.3 Summary

This chapter presented an overview of the functionality provided by SeeIT 3D. It can be classified in IDE interactions and visualization interactions. The former offer the user a set of controls immersed within the IDE functionality, while the latter change the state of the visualization allowing him to modify it as he wants.

Chapter 5

SeeIT 3D to Understand Open Source Software Projects

The main objective of SeeIT 3D is to visualize software projects written in Java (although it is easily extensible to other languages by using the XML based visualization). In order to demonstrate how the tool performs when visualizing real data, there were chosen 5 open source projects. These projects are: aTunes, DrJava, JFreechart, Spring Framework core and Wicket. All of them are available in the Internet and well known because of their popularity inside the field they represent. In the following sections will be illustrated how these projects "look" by using the SeeIT 3D metaphor, and how it can help developers to find problems or emerging patterns in the code.

In each project, there were chosen three views of the system using the Blue to Yellow color scale: a general overview at package level, a general overview at class level and a view showing the relationships at package level. With these three views is possible to get a general idea of the quality of the system by taking a quick look at some of the most important characteristics represented by the software metrics included.

The first view shows a general overview at package level, this means that each polycylinder represents a whole package. In all the examples the mapping between visual properties and metrics has been established as: *height* of the polycylinder representing the complexity of the package and the number of lines of code is represented by a color scale[1].

Increasing the granularity level helps to get wider knowledge of the system. This change in the granularity level leads to the second figure of each project. It depicts a general overview of the project, but in this case each polycylinder represents a class contained within the project. At this level an additional metric is applied, the Lack of Cohesion. By using three metrics is possible to map every visual property to a metric, allowing to get more information about the software system. In each corresponding figure, the mapping was setted to: *height* represents the complexity of the class, *color* indicates the number of lines of code and the *width* represents the lack of cohesion. By using this view is possible to identify classes with higher values than others or even small clusters of polycylinders with common values. Almost in every case the clusters appear within the same package because polycylinders are placed one after the other when they belong to the same package.

[1]One different for each analyzed project

The third view allows the user to understand how the artifacts are related to each other. Following the relationships is possible to identify relationships among artifacts with bad metrics which is the first step to fix the problems encountered.

The result of the analysis of each project is described in the following sections.

5.1 aTunes

What is it?: "aTunes is a full-featured audio player and manager, developed in Java programming language, so it can be executed on different platforms: Windows, Linux and Unix-like systems. Currently it plays mp3, ogg, wma, wav, flac, mp4 and radio streaming, allowing users to easily edit tags, organize music and rip Audio Cd's."[2]

aTunes has history from late 2006; it currently has 117 packages, 809 classes, 5004 methods, 30649 lines of code within methods and 57113 lines of code in total. It provides a wide variety of features allowing the users to count with a well written and complete audio player as stated in the description.

5.1.1 Visual Analysis

Figure 5.1 depicts the general overview at package level. This figure highlights that aTunes packages have a high number of lines of code (yellow polycylinders).

Figure 5.2 shows the general overview at class level and in figure 5.3 can be seen an example of visualization where the relationships (represented by arcs) are rendered.

5.2 DrJava

What is it?: "DrJava is a lightweight development environment for writing Java programs. It is designed primarily for students, providing an intuitive interface and the ability to interactively evaluate Java code. It also includes powerful features for more advanced users. DrJava is available for free under the BSD License, and it is under active development by the JavaPLT group at Rice University."[3]

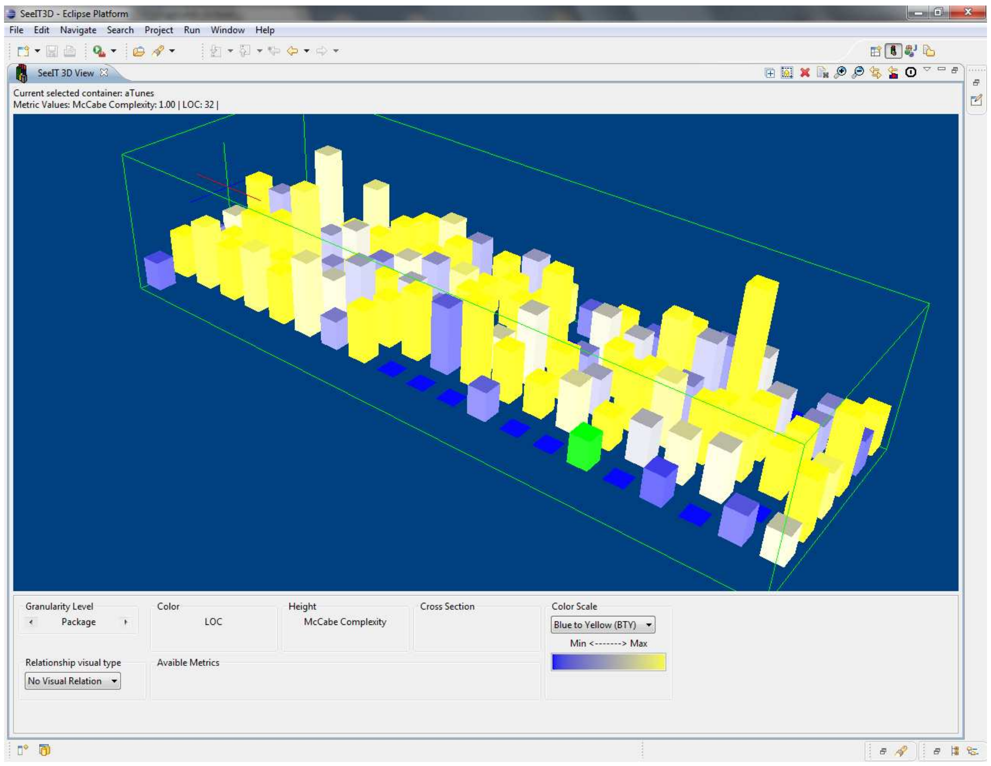
DrJava has history from 2001; it currently has 29 packages, 782 classes, 7137 methods, 63683 lines of code within methods and 89021 lines of code in total.

5.2.1 Visual Analysis

Figure 5.4 shows the general overview at package level. In this case it is possible to see that most of the packages have a high number of lines of code (yellow in the color scale).

Figure 5.5 depicts the general overview at class level while figure 5.6 illustrates the relationships view of DrJava. In this later view is possible to observe that the package *edu.rice.cs.drjava* have 12 classes but is

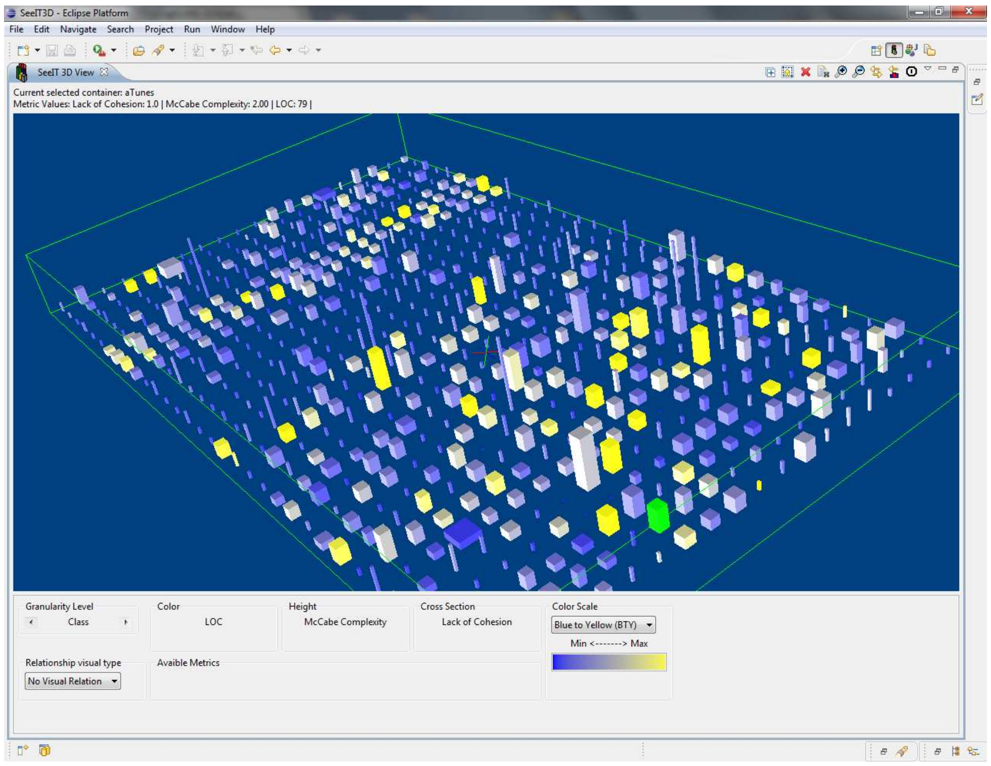
[2]http://www.atunes.org/
[3]http://drjava.org/


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: aTunes
Metric Values: McCabe Complexity: 1.00 | LOC: 32 |

Granularity Level: Package
Color: LOC
Height: McCabe Complexity
Cross Section
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

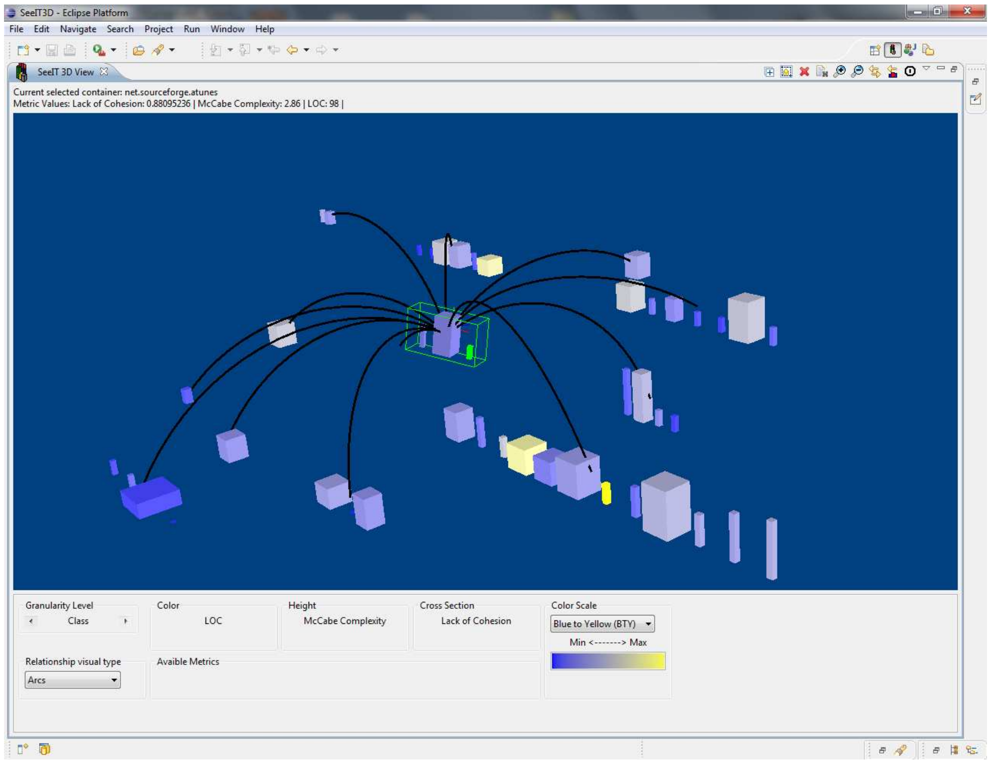
Figure 5.1: aTunes overview at package level


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: aTunes
Metric Values: Lack of Cohesion: 1.0 | McCabe Complexity: 2.00 | LOC: 79 |

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

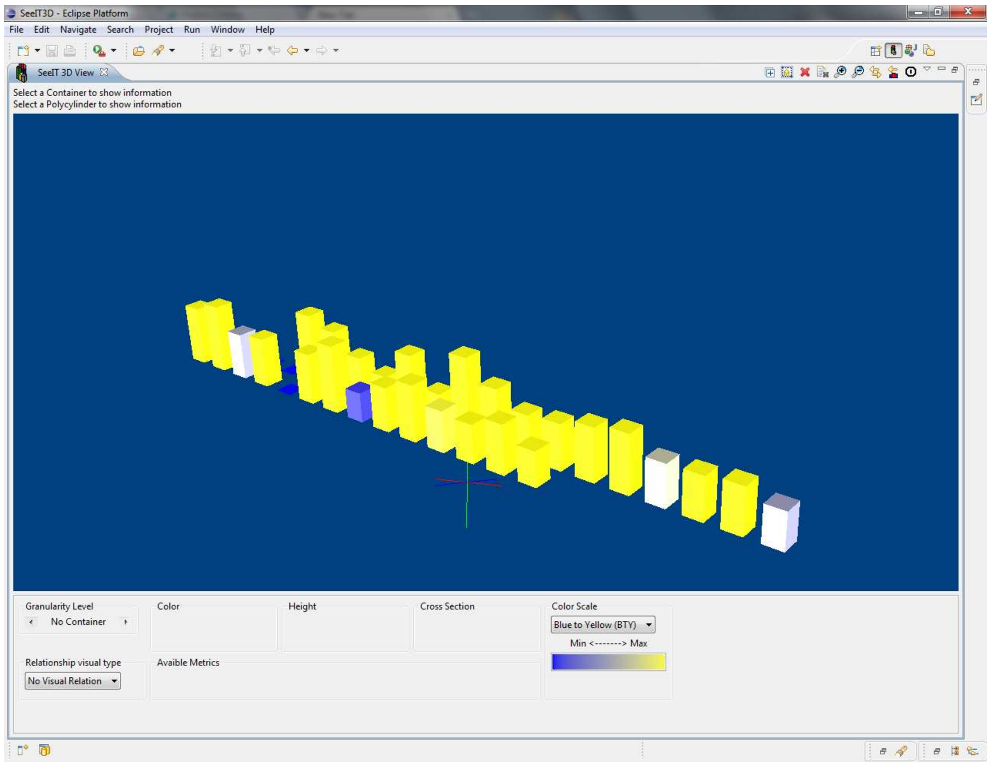
Figure 5.2: aTunes overview at class level


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: net.sourceforge.atunes
Metric Values: Lack of Cohesion: 0.88095236 | McCabe Complexity: 2.86 | LOC: 98 |

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: Arcs
Avaible Metrics

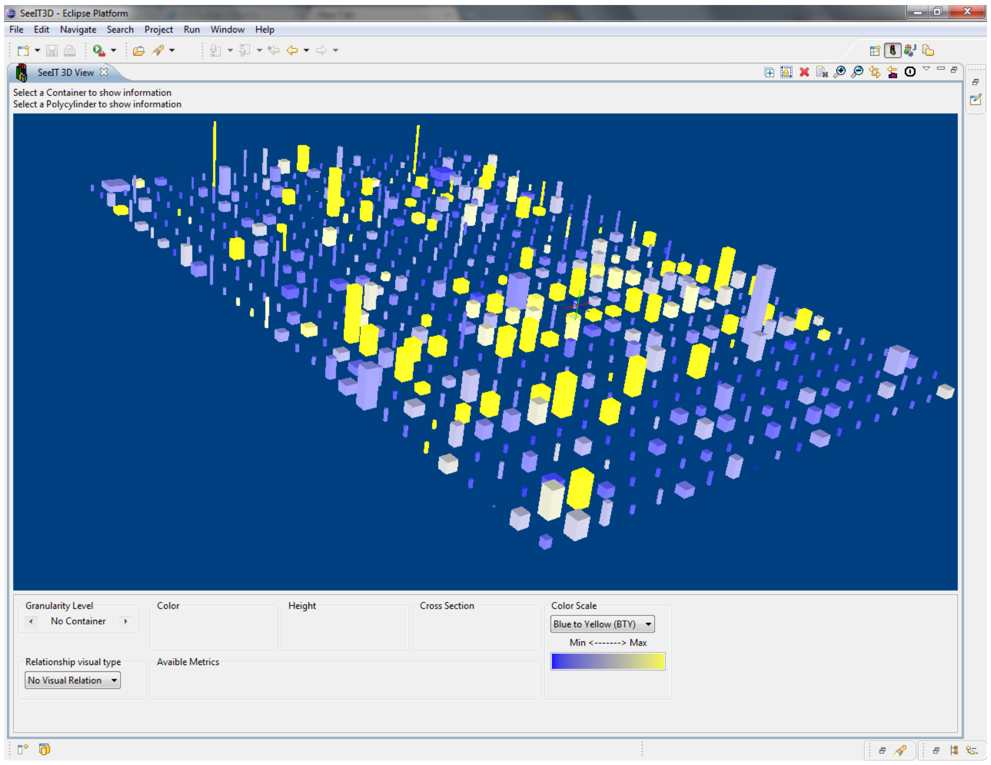
Figure 5.3: Relationships among aTunes artifacts


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Select a Container to show information
Select a Polycylinder to show information

Granularity Level: No Container
Color
Height
Cross Section
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

Figure 5.4: DrJava overview at package level


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View
Color
Height
Cross Section
Min <-------> Max

Select a Container to show information
Select a Polycylinder to show information

Granularity Level: No Container
Color Scale: Blue to Yellow (BTY)
Relationship visual type: No Visual Relation
Avaible Metrics

Figure 5.5: DrJava overview at class level

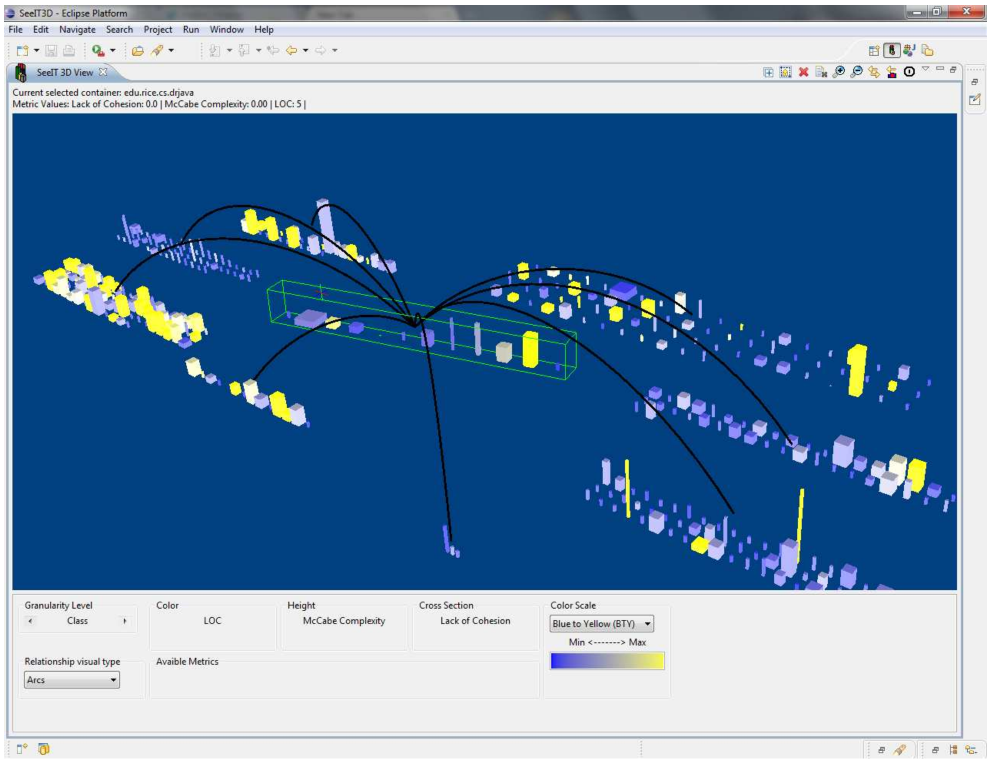
related to 8 other packages indicating that the evaluated package uses a wide variety of other functionality represented by other packages. This could be seen as a smell because the evaluated package is small but requires much information from other places.

5.3 JFreechart

What is it?: "JFreeChart is a free 100% Java chart library that makes it easy for developers to display professional quality charts in their applications. JFreeChart's extensive feature set includes: a consistent and well-documented API, supporting a wide range of chart types; a flexible design that is easy to extend, and targets both server-side and client-side applications; support for many output types, including Swing components, image files (including PNG and JPEG), and vector graphics file formats (including PDF, EPS and SVG). JFreeChart is "open source" or, more specifically, free software. It is distributed under the terms of the GNU Lesser General Public License (LGPL), which permits use in proprietary applications."[4]

JFreechart has history from 2000; it currently has 37 packages, 504 classes, 7551 methods, 73615 lines of code within methods and 91174 lines of code in total. This is the biggest project chosen, at least in terms of lines of code. Two committers participate in this project; it can be seen as a bad practice as there may not be conceptual differences between them. This could easily lead to code with bad practices since there is no one to correct them.

[4] http://www.jfree.org/jfreechart/


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: edu.rice.cs.drjava
Metric Values: Lack of Cohesion: 0.0 | McCabe Complexity: 0.00 | LOC: 5 |

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: Arcs
Avaible Metrics

Figure 5.6: Relationships among DrJava artifacts

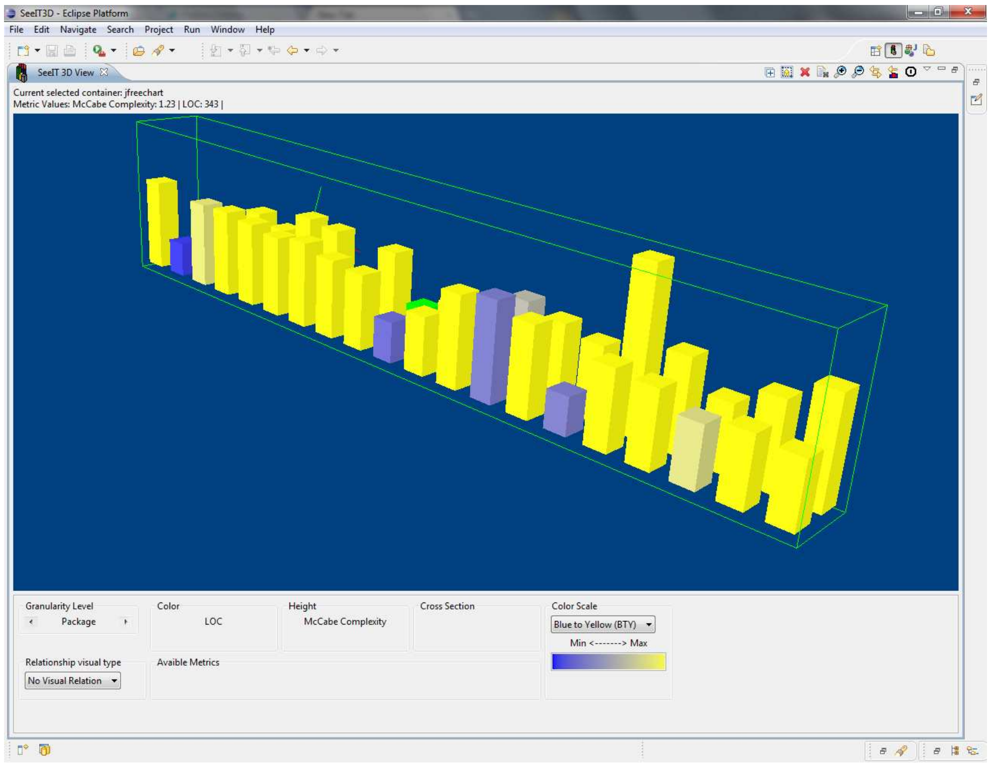
5.3.1 Visual Analysis

Figure 5.7 shows the general overview at package level of JFreechart. Once again, in this view can be seen a high number of lines of code, and a medium value of complexity compared with the other projects because the polycylinders representing the packages are taller than the others.

Figure 5.8 present a view of the project overview at class level. A notorious characteristic shown by this view is the lack of cohesion among the classes of this system, since the scene is much more dense than the others, which indicates that they should be split to reach a better internal architecture.

Finally, figure 5.9 shows the relationships view of the package *org.jfree.chart.* It has a high number of related packages, considerable size in terms of lines of code, high complexity and high lack of cohesion. This package needs to be corrected due to its bad metric values.

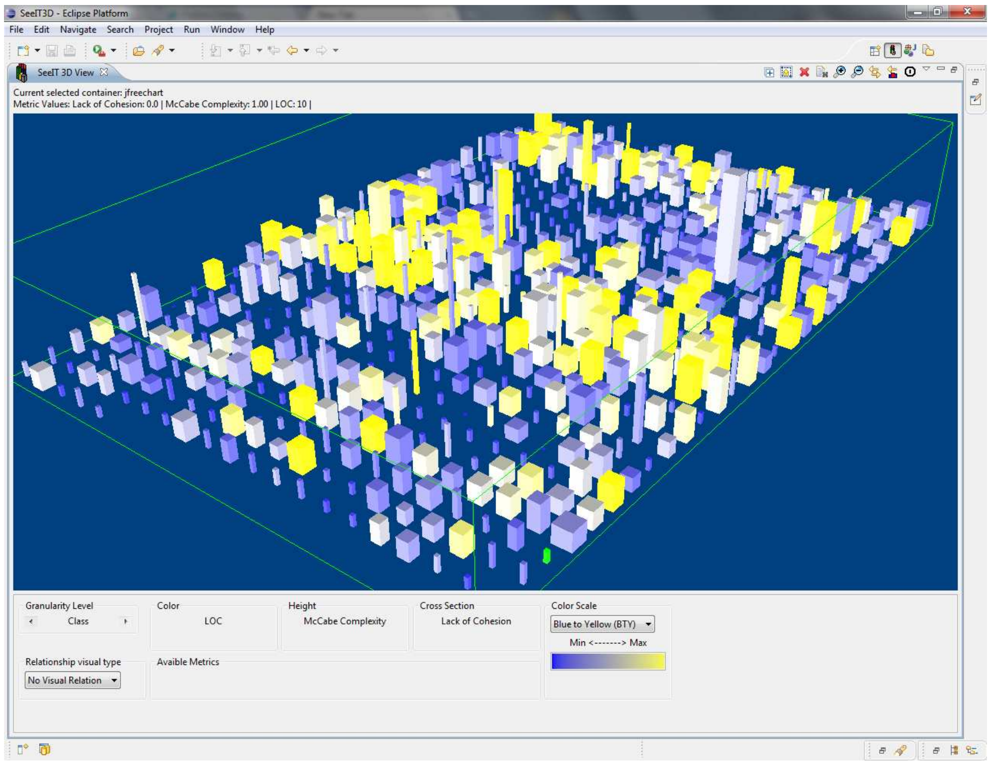
Because of the characteristics found in this project, a deeper analysis was conducted. It was chosen the class with the highest complexity. This class was rendered using the lowest granularity level possible, the line. At this granularity level it is applied the *Control Structure* metric which allows seeing the control structures used in the code. Figure 5.10 shows the view produced by this configuration. This view highlights that the class contains a high number of *if* statements, represented in green. That is the reason for the high value of the McCabe complexity.


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: jfreechart
Metric Values: McCabe Complexity: 1.23 | LOC: 343 |

Granularity Level: Package
Color: LOC
Height: McCabe Complexity
Cross Section
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

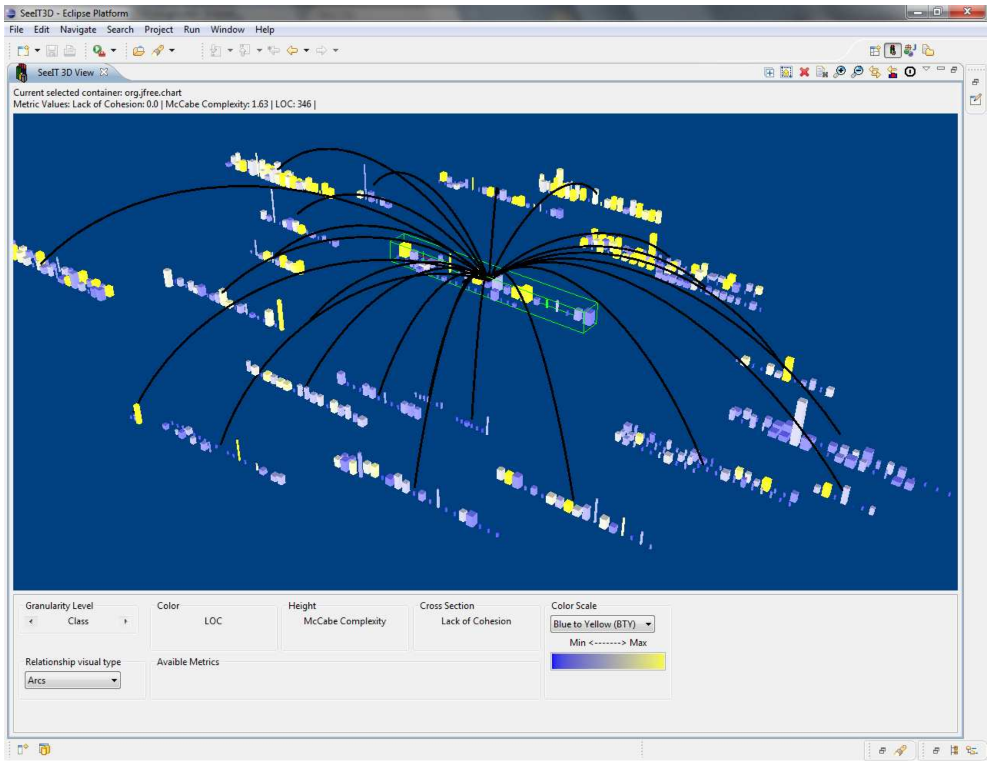
Figure 5.7: JFreechart overview at package level


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: jfreechart
Metric Values: Lack of Cohesion: 0.0 | McCabe Complexity: 1.00 | LOC: 10 |

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

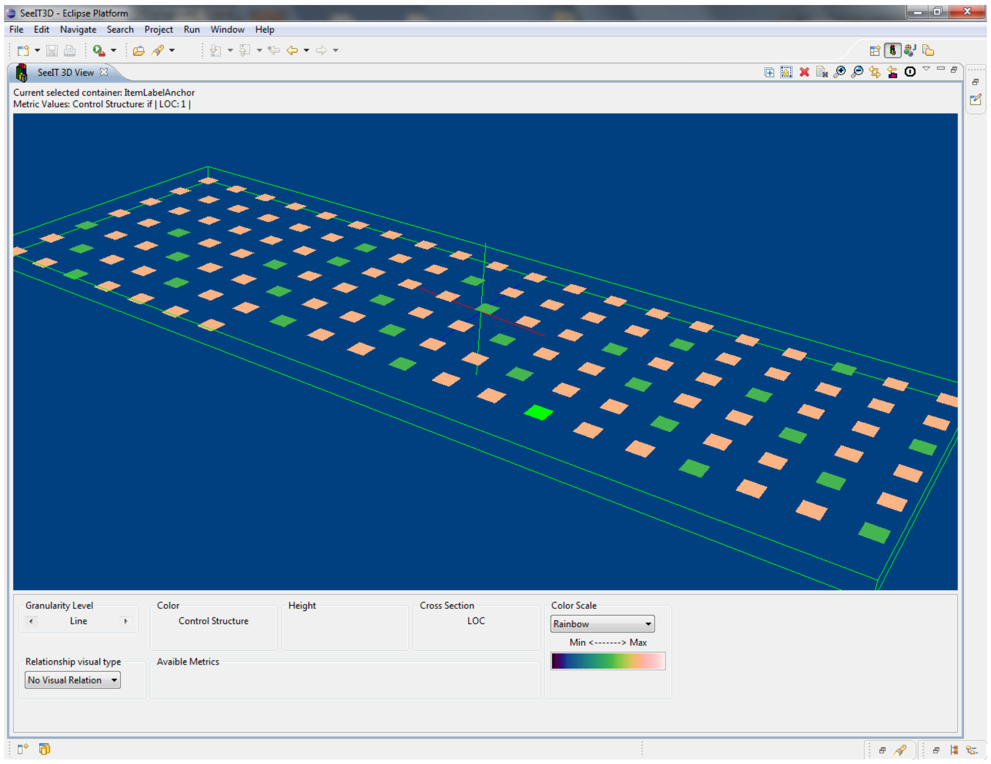
Figure 5.8: JFreechart overview at class level


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: org.jfree.chart
Metric Values: Lack of Cohesion: 0.0 | McCabe Complexity: 1.63 | LOC: 346 |

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: Arcs
Avaible Metrics

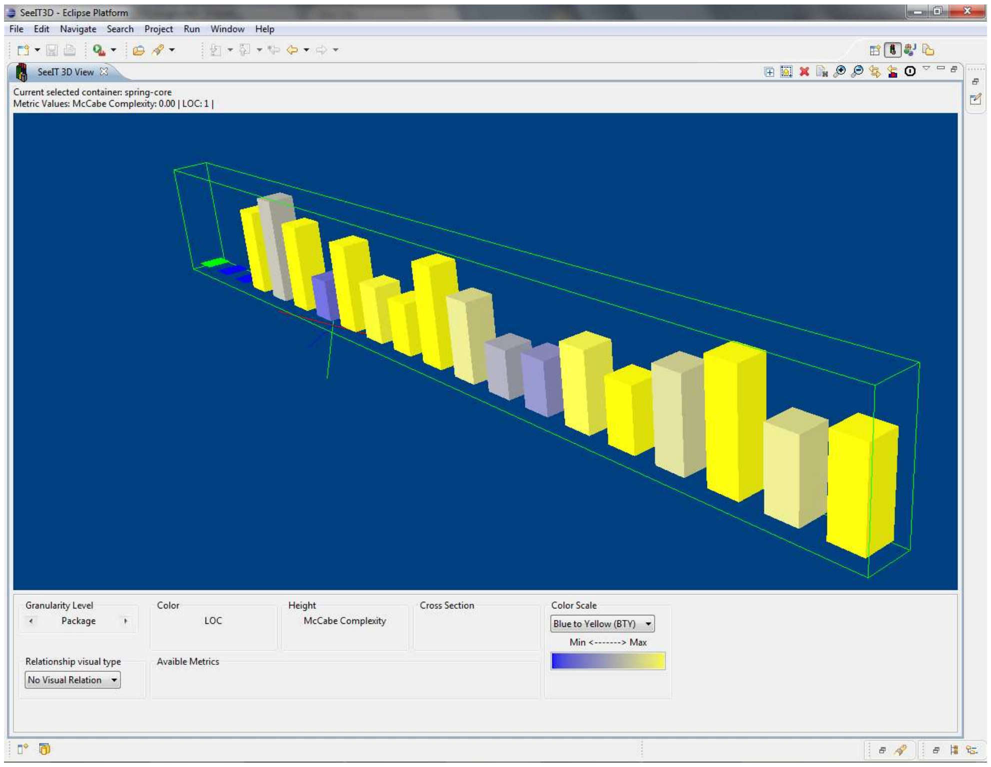
Figure 5.9: Relationships among JFreechart artifacts


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: ItemLabelAnchor
Metric Values: Control Structure: if | LOC: 1 |

Granularity Level: Line
Color: Control Structure
Height
Cross Section: LOC
Color Scale: Rainbow
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

Figure 5.10: JFreechart highest complexity artifact


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: spring-core
Metric Values: McCabe Complexity: 0.00 | LOC: 1 |

Granularity Level: Package
Color: LOC
Height: McCabe Complexity
Cross Section
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

Figure 5.11: Spring core overview at package level

5.4 Spring Core

What is it?: “Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

Spring enables you to build applications from “plain old Java objects” (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE...”[5]

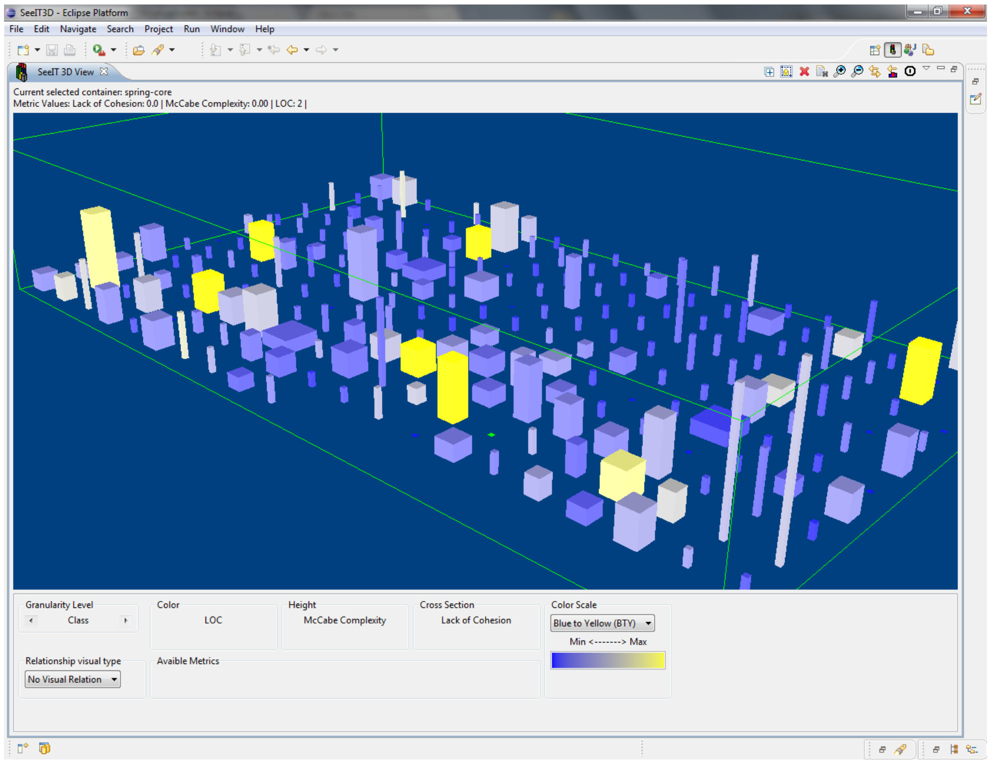
Spring has history from 2002; it currently has 17 packages, 206 classes, 1119 methods, 9049 lines of code within methods and 14443 lines of code in total. The analyzed module represents a small part of the bigger Spring framework project. The analysis was performed over one of the fundamental pieces of it, its core, as such it performs important operations and therefore it needs be well constructed.

5.4.1 Visual Analysis

In figure 5.11 can be seen the package level overview that allows us to see that the project has a small size. This view highlights the mentioned size of the project by showing a low value in about half of the packages.

Next on figure 5.12 the overview at class level is presented. This view presents a container with low density of polycylinders, indicating that the classes have a low level of lack of cohesion, as well as few polycylinders with high values of complexity (represented by the height).

[5]http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/overview.html


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: spring-core
Metric Values: Lack of Cohesion: 0.0 | McCabe Complexity: 0.00 | LOC: 2 |

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

Figure 5.12: Spring core overview at class level

Finally, figure 5.13 depicts the package *org.springframework.core.io*. It only has two related packages but it is a package with a high number of relationships compared to others. This package is an example of the structured way this project is built, by making few dependencies among packages.

5.5 Wicket

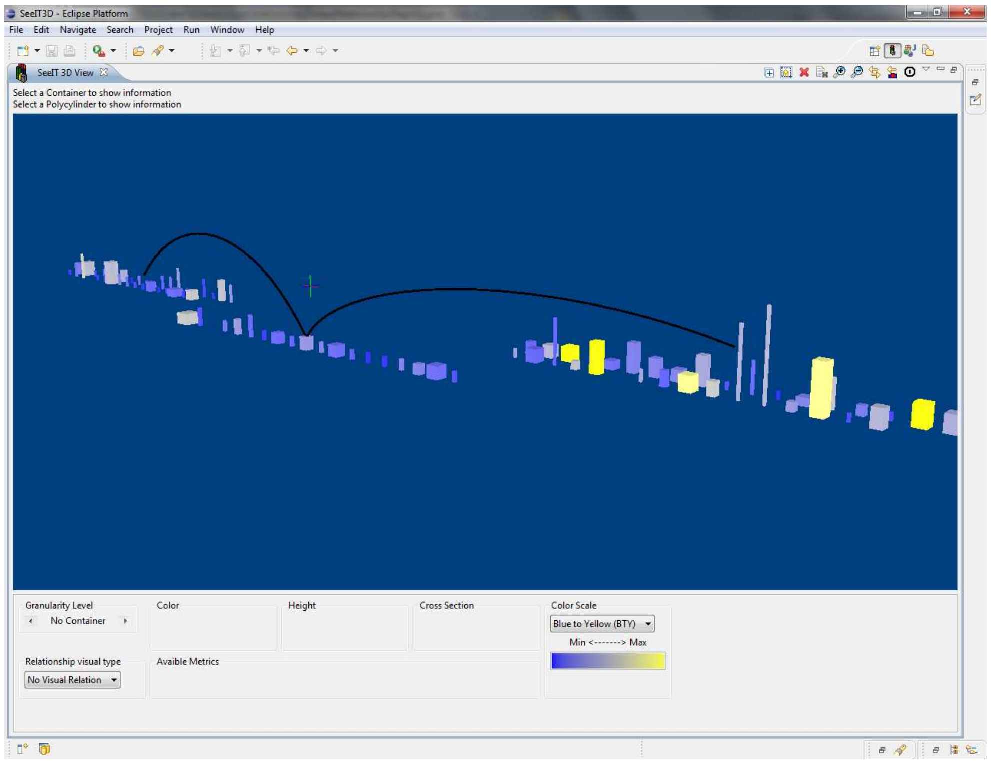
What is it?: "With proper mark-up/logic separation, a POJO data model, and a refreshing lack of XML, Apache Wicket makes developing web-apps simple and enjoyable again. Swap the boilerplate, complex debugging and brittle code for powerful, reusable components written with plain Java and HTML."[6]

Wicket has history from 2004; it currently has 95 packages, 867 classes, 6676 methods, 50172 lines of code within methods and 83273 lines of code in total. This project is focused on the good use of OO principles that allow building software in a maintainable and well structured manner.

5.5.1 Analysis

First, in figure 5.14 is shown the general overview at package level, where once again it can be analyzed the size, and general characteristics of the project. It is possible to observe that Wicket is a big project composed by several packages with a high number of lines of code and medium complexity at package level.

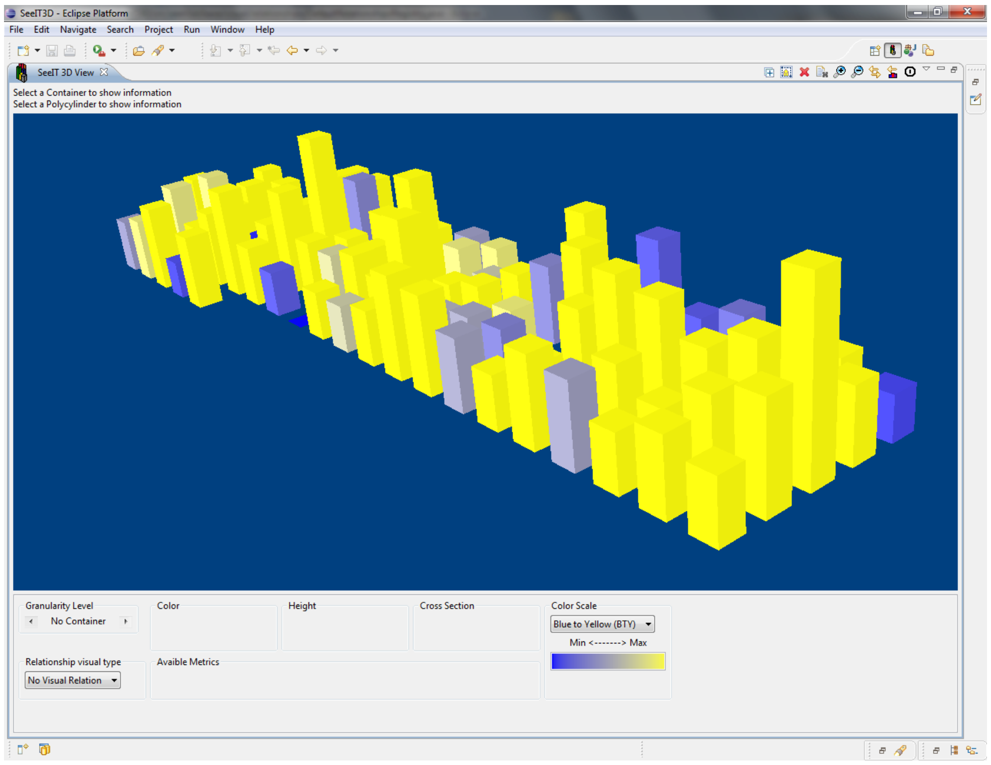
[6] http://wicket.apache.org/


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View
Color
Height
Cross Section
Avaible Metrics
Min <-------> Max

Select a Container to show information
Select a Polycylinder to show information

Granularity Level: No Container
Relationship visual type: No Visual Relation
Color Scale: Blue to Yellow (BTY)

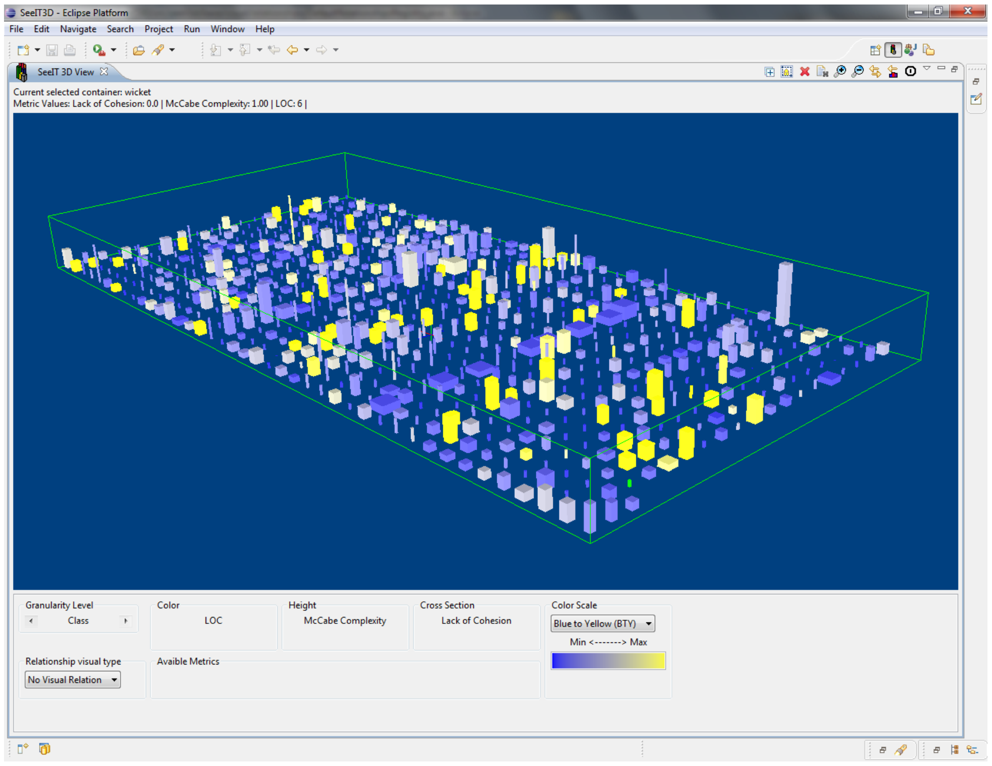
Figure 5.13: Relationships among Spring core artifacts


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View
Color
Height
Cross Section
Avaible Metrics
Min <-------> Max

Select a Container to show information
Select a Polycylinder to show information

Granularity Level: No Container
Relationship visual type: No Visual Relation
Color Scale: Blue to Yellow (BTY)

Figure 5.14: Wicket overview at package level


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: wicket
Metric Values: Lack of Cohesion: 0.0 | McCabe Complexity: 1.00 | LOC: 6 |

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

Figure 5.15: Wicket overview at class level

Second, in figure 5.15 is presented the overview at class level. This view renders a lot of information because of the number of classes within the project. Despite the number of classes, most of them have low levels in metric values with some exceptions like the widest and lowest polycylinders that are singleton classes with one property that is not used by any of the class's methods[7].

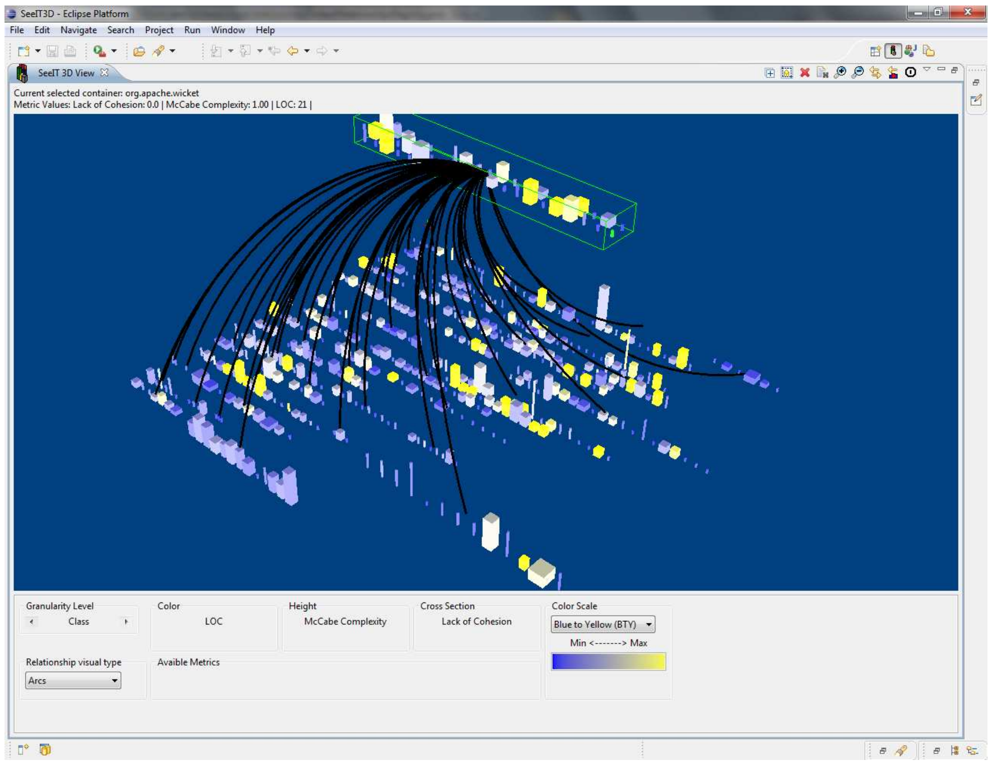
Third, in figure 5.16 the related packages of the evaluated artifact *org.apache.wicket* can be observed. There are a high number of related containers indicating that the functionality of the package should be divided into other packages because it is possible that this package contains incorrectly classified classes.

Finally, in figure 5.17 the same package and its related containers is shown, but the relationships are represented using the common base relationship generator. This way the great amount of lines in the visualization is removed.

5.6 Comparing five projects

In figures 5.18 and 5.19 the five projects are rendered under the same view. In the first figure the mapping was setted to: *color* represents the number of lines of code, *height* means complexity and *width* highlights the lack of cohesion. *Containers* on the other side, represent each project where each polycylinder is a class of the corresponding project. In the second figure the mapping was changed to: *color* is lack of cohesion, *height* is number of lines of code and *width* is complexity.

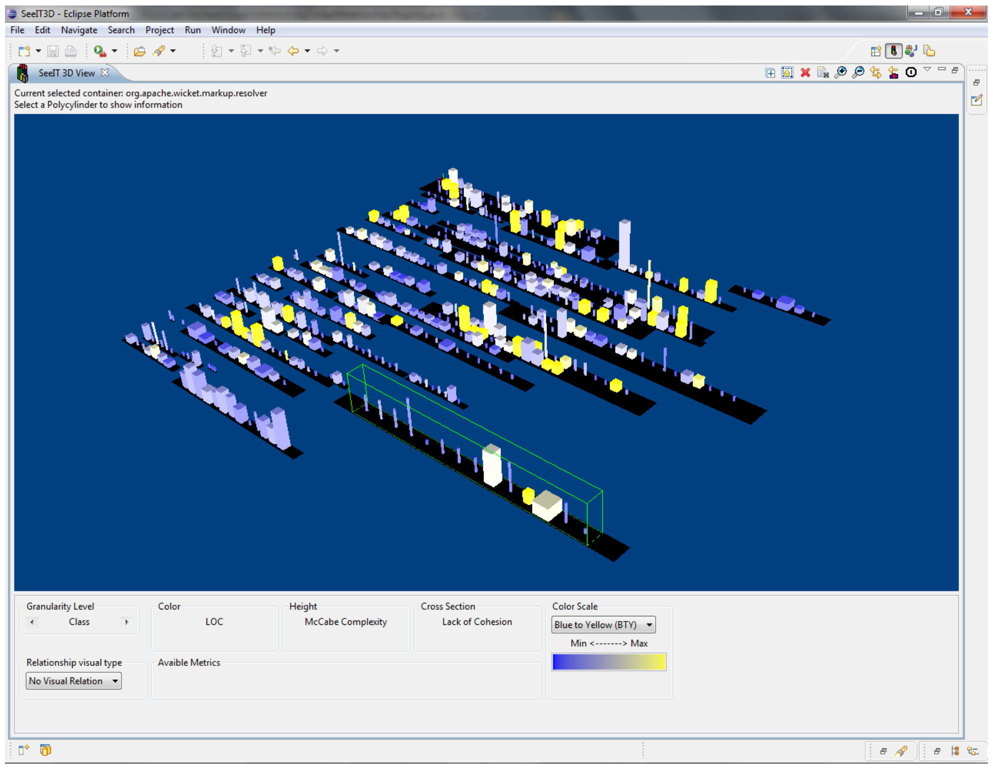
[7]This verification was performed by using the link between polycylinder and artifact in the package explorer view of Eclipse


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: org.apache.wicket
Metric Values: Lack of Cohesion: 0.0 | McCabe Complexity: 1.00 | LOC: 21 |

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: Arcs
Avaible Metrics

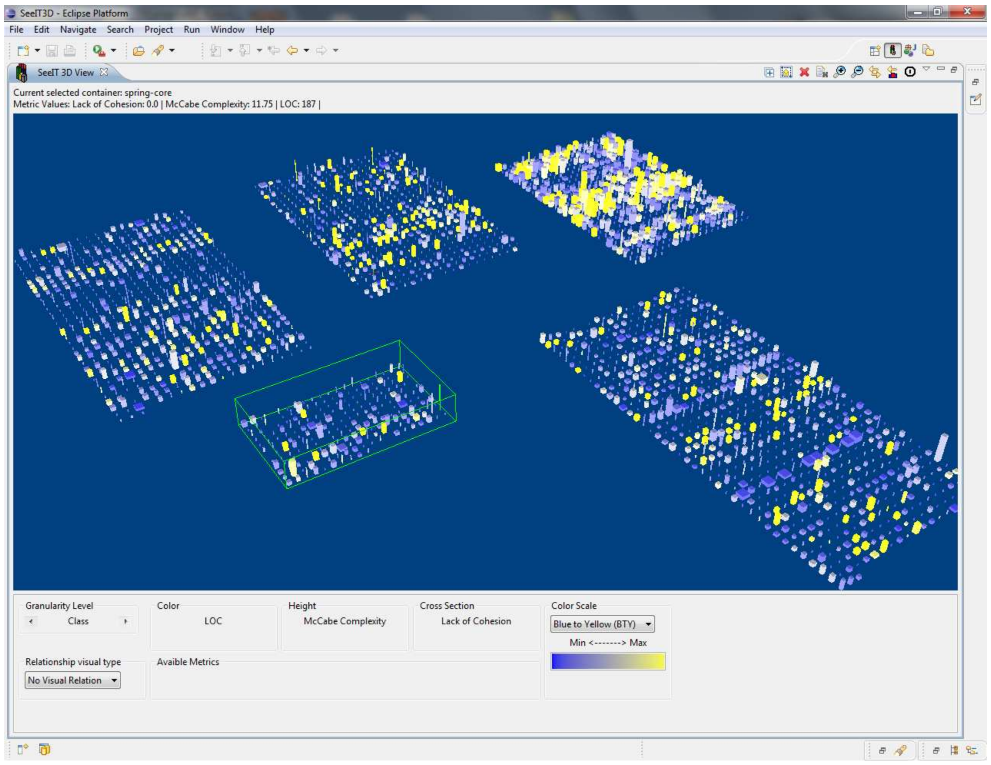
Figure 5.16: Relationships among Wicket artifacts (using arcs)


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: org.apache.wicket.markup.resolver
Select a Polycylinder to show information

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

Figure 5.17: Relationships among Wicket artifacts (using common base)


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: spring-core
Metric Values: Lack of Cohesion: 0.0 | McCabe Complexity: 11.75 | LOC: 187 |

Granularity Level: Class
Color: LOC
Height: McCabe Complexity
Cross Section: Lack of Cohesion
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics

Figure 5.18: Side by side overview of five projects

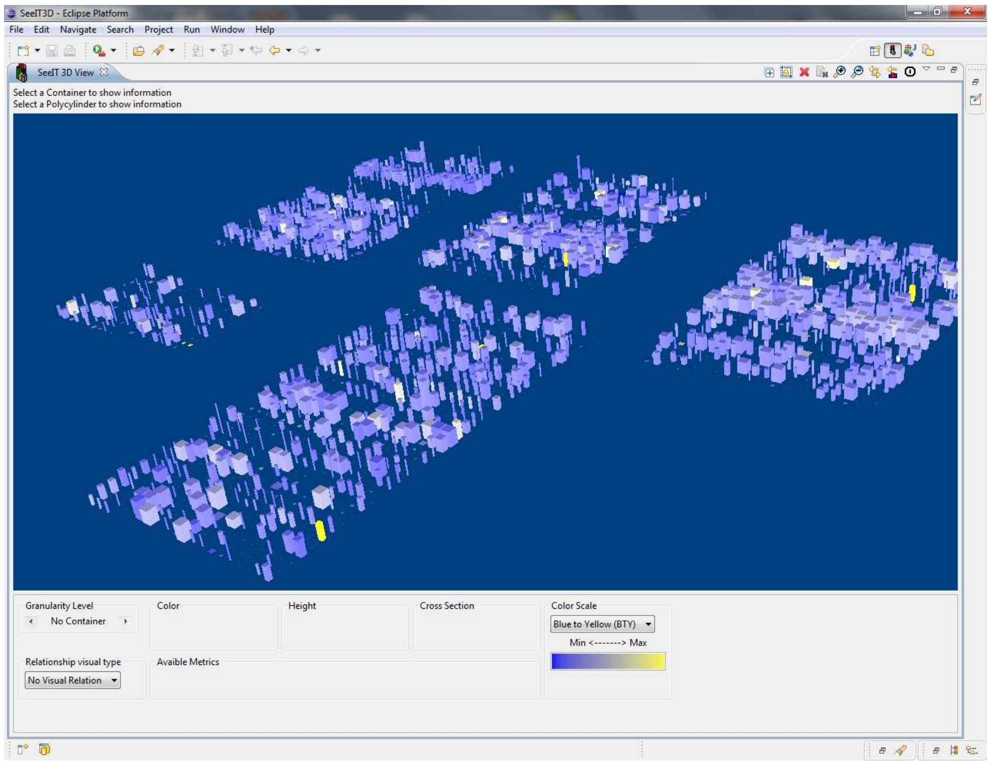
In this pair of figures is possible to observe the previously stated characteristics, except that they are easier to highlight and compare with each other, since they are all within the same scene.

5.7 Visualizing a Relational Database

SeeIT 3D offers several options to visualize software artifacts. Until now the tool has been tested with software projects within the IDE. This section presents how the XML based visualization can be used to visualize other kinds of artifacts like a relational database.

In figure 5.20 is shown an example of a relational database used by a university to manage its students and courses they are taking. The same database is viewed with SeeIT 3D and presented in figure 5.21.

In order to build these visualization, it was developed a small program to extract information about the database and translate it to the XML format specified by SeeIT 3D. This program analyzed the database looking for tables, data types within these tables and the number of rows each table had. With this information the tool creates a representation where each table is represented by a *container* and each column of the table is a corresponding *polycylinder*. The relationships among tables (foreign keys) are represented using arcs to connect the containers. The data type of the column and the number of rows in each table are taken as metrics. This representation allowed to build a view of the database like the one presented in figure 5.21. It is relatively easy to notice that the predominant data type in the visualization are textual values (expressed in light blue color), and that the table "grade" have the highest number of rows as it is the container with the highest polycylinders.


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View
Color
Height
Cross Section
Avaible Metrics
Min <-------> Max

Select a Container to show information
Select a Polycylinder to show information

Granularity Level: No Container
Color Scale: Blue to Yellow (BTY)
Relationship visual type: No Visual Relation

Figure 5.19: Side by side five project mapping changed

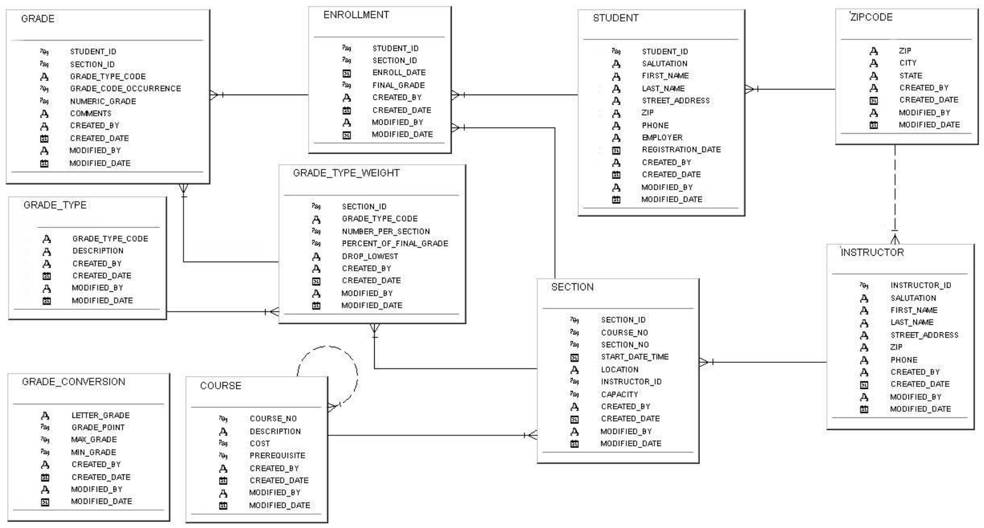
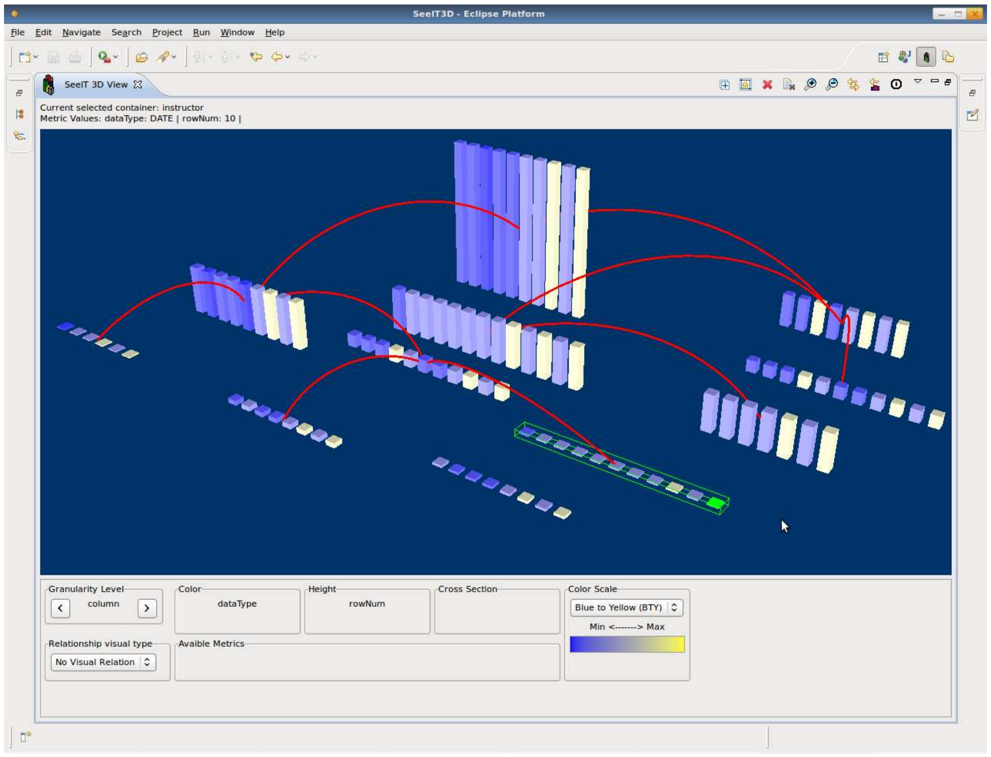


Figure 5.20: Relational database example


SeeIT3D - Eclipse Platform
File Edit Navigate Search Project Run Window Help
SeeIT 3D View

Current selected container: instructor
Metric Values: dataType: DATE | rowNum: 10 |

Granularity Level: column
Color: dataType
Height: rowNum
Cross Section:
Color Scale: Blue to Yellow (BTY)
Min <-------> Max
Relationship visual type: No Visual Relation
Avaible Metrics:

Figure 5.21: Relational database viewed with SeeIT 3D

The above mentioned conclusions cannot be extracted easily when looking at the classic relational database modeling.

5.8 General Conclusions

When using SeeIT 3D to visualize these projects, it was possible to find some interesting spots in the visualization. First, by mapping the lack of cohesion metric to the width of each polycylinder and visualizing at class level, it was possible to identify utility classes because they usually have a high value in the lack of cohesion metric and low value in the complexity metric that leads to a low-height and high-width polycylinder.

Second, by taking a quick look at the polycylinders' width when viewing a project at class level, it was possible to identify which classes had high values in the lack of cohesion metric, it indicates that these classes should be splitted into two or more classes. In the same way, it also was possible to identify the most complex classes and packages by simply looking at the highest polycylinders within a certain view.

Third, the relationships density indicated that some of these projects should be refactored to support better separation of concerns among packages.

Fourth, the possibility of visually compare a set of projects, allows developers to fix design errors. When they compare their system design (expressed by the view) with other, is possible to learn from others. It is an important advantage that can help developers to write better software systems for their clients.

Fifth, the emerging patterns are not limited by the ones highlighted by the author. It would be an endless task to find every pattern because of the intrinsic characteristics of each project and the multiple configurations that can be expressed with the metaphor. Instead, a user can find new visualization patterns based on his experience with the tool.

Finally, it is worth mentioning how the metaphor can be used to visualize different kinds of information like the database example seen in figure 5.20. It allowed us to easily identify how the tables are related to each other and what was the most used data type by the system using this database.

5.9 Summary

This chapter presented the wide spectrum of possibilities to visualize with SeeIT 3D. There were analyzed five open source projects written in Java. They included: aTunes, DrJava, JFreechart, Spring Core and Wicket. Within the analysis, it was shown how the visualization allowed us to find patterns and helped to determine the quality of the software written in each of these projects. Then, it was presented how SeeIT 3D can be used in conjunction with other tools to visualize information from different sources as well as different artifacts. Finally, several conclusions about the use of the tool were discussed.

Chapter 6

Tool Evaluation

The evaluation of SeeIT 3D can be done with two different techniques. The first one compares it with similar tools found in Eclipse Marketplace[1]. The second one uses an evaluation framework to do a less subjective analysis.

The first approach allows us to find out if the interaction level with the IDE is the appropriated, determine missing functionalities, compare similar features, and assess the importance and usefulness of novel functions. Besides, it also opens the possibility to evaluate certain aspects like performance, usability and quality of the information.

The second approach is based on the work done by software visualization researchers. In general, an evaluation framework highlights the most important aspects that need to be considered when you are evaluating a software visualization tool. So these frameworks offer a wide and organized list of factors the evaluator needs to consider, while he is assessing a tool.

These two approaches were combined to illustrate what SeeIT 3D is able to do. The results suggested the tool offers a suitable set of features and operations which are ready to be used by developers working on a real Java project. However, before this, an extrinsic evaluation should be carried out. By using this type of evaluations researchers analyze the performance of two groups, A and B, doing the same development or maintenance task, ideally, under identical conditions. Team A uses the evaluated tool and B does not. As results of the experiment, an assessment of the positive and negative influences of the tool on a development team, and the opinions, thoughts and suggestions given by participants are the most valuable outcomes. The design and implementation of this sort of experiment is one of the most important items of the future work on this research.

6.1 Comparing with Other Similar Tools

Two similar tools were selected, X-Ray [27] and Codstruction. These are visualization tools embedded within the Eclipse IDE and provide data analysis over the source code of a project. X-Ray and Codstruction are presented in the next subsections, as well as a comparison table of both tools versus SeeIT 3D.

[1]Eclipse Marketplace is a plugin repository for the Eclipse IDE. It can be found at http://marketplace.eclipse.org/

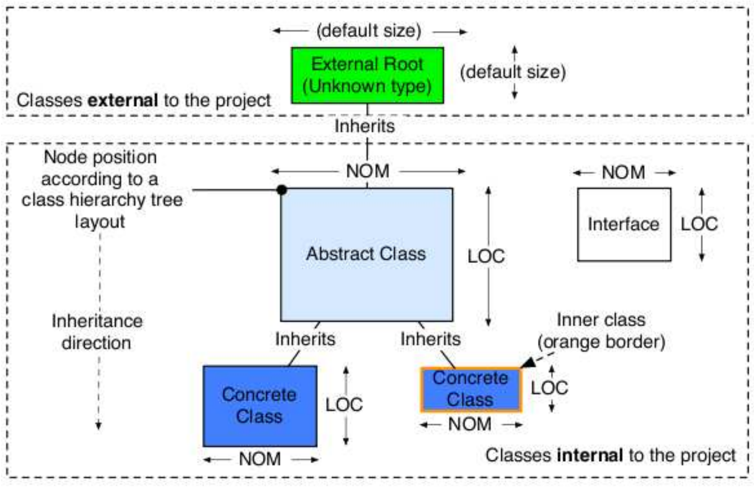

(default size)
External Root (Unknown type)
(default size)
Classes **external** to the project
Inherits
Node position according to a class hierarchy tree layout
NOM
NOM
Abstract Class
LOC
Interface
LOC
Inheritance direction
Inherits
Inherits
Inner class (orange border)
Concrete Class
LOC
Concrete Class
LOC
NOM
NOM
Classes **internal** to the project

Figure 6.1: X-Ray metaphor concept[27]

6.1.1 X-Ray

X-Ray makes use of a bidimensional view based on rectangles to show hierarchical and dependency relationships among classes. Its metaphor can be seen in figure 6.1. It uses three metrics to represent information: Number of Methods (NOM), Lines of Code (LOC) and type of the class (namely concrete class, abstract class, interface and external class). Each of these metrics is mapped to a visual property: NOM to width, LOC to height and type of the class to color of each rectangle. Other type of view offered by X-Ray is the class and package dependency view; it allows visualizing the dependencies among classes and packages. It places the analyzed artifacts (classes or packages) in a circle and then, it draws lines among them to represent their interdependencies. The line thickness indicates the weight of the relation i.e. the thicker the line the more coupled the artifacts are. An example of this view can be seen in figure 6.2.

The use of two dimensional views and graph based representations makes difficult to scale the metaphor when viewing large software projects. Although X-Ray allows highlighting information segments, it is not easy to navigate using the mouse buttons or keyboard.

X-Ray allows navigating from the view to code by using a link between the visualization and the package explorer view of the IDE.

In figure 6.3 is shown the visualization of the JFreechart project. This view demonstrates that with the amount of information produced by the analysis makes difficult to get a general overview of the system.

6.1.2 Codstruction

Codstruction is based on the CodeCity metaphor [42]. It uses a metaphor based on a real world representation: a city. This type of metaphor was created with the purpose to provide a sense of locality. The lack of locality and navigation issues in 3D visualization environments has been criticized as stated in [42], so this metaphor is focused in this aspect.

The CodeCity metaphor defines its visual elements as: classes are buildings, packages are city districts, the Number of Methods (NOM) is the height of the building that represents the class, and the Number

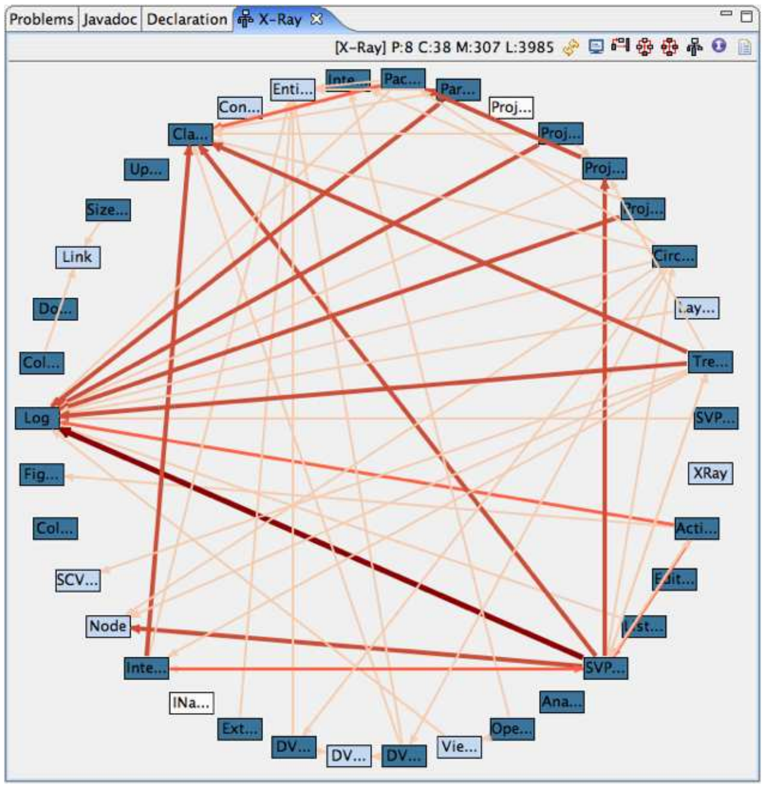


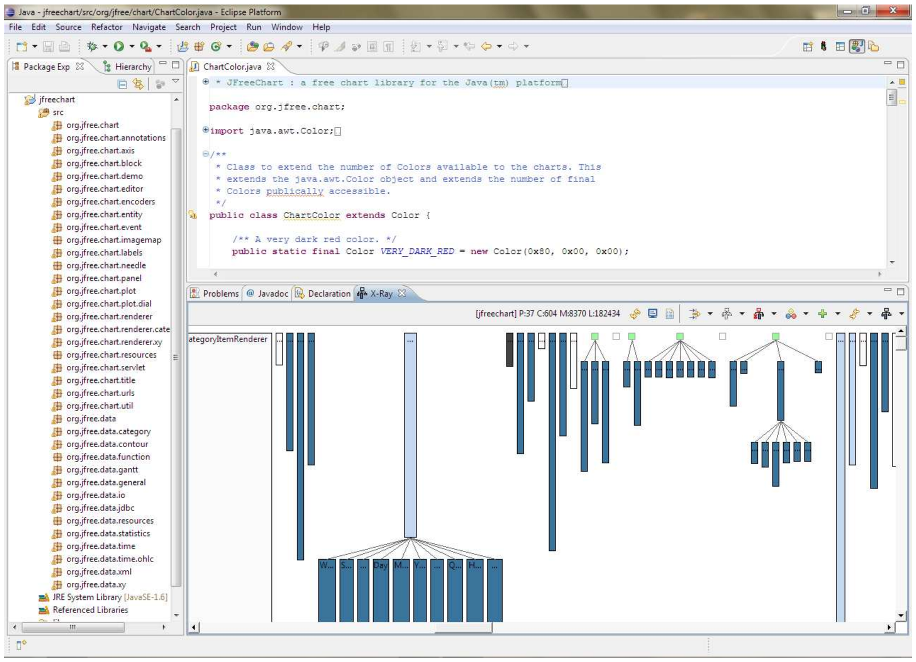
Figure 6.2: Class and Package dependency view

of attributes (NOA) of the class corresponds to the width of it. Using this mapping, the authors tried to indicate that high buildings belong to business in a real city, while in the visualization of the software they represent classes with high functionality (business logic). They also indicate that using this kind of visualization it is possible to detect *brain classes*, *god classes* and *data classes* by using a visualization based on a disease map representation. Concerning the navigation issues, CodeCity's metaphor uses two movement axis, *vertical* (parallel to buildings) and *horizontal* (parallel to ground) which enable the user to navigate the world as in a real city. The definition of the mapping was established by the authors according to a set of experiments, unfortunatetly this mapping can not be changed as the user wants, forcing him to see the software system from a fixed point of view and granylarity level. The original version of CodeCity posses a *query* mechanism that allows the user to specify pieces of information related to the software system; along with this query mechanism is located a tagging system that allows to store extra information that may be important for the user. These two last characteristics are not included in the Codstruction plugin.

Codstruction extends the CodeCity metaphor by stacking buildings one over the other to indicate hierarchical relationships among classes. Other key difference is the way interfaces as rendered; in the case of CodeCity they are represented as buildings, while in Codstruction they are represented using pyramids.

Codstruction allows the interaction between the visualization and the analyzed data. It makes use of the editor when a building is double clicked. Despite this link between the code and visualization, it does not have a filtering method or an information selection method which limits the visualization to a fixed view.

Figure 6.4 presents a view of the JFreeChart project using this tool. It shows the whole city representing



```
 * JFreeChart : a free chart library for the Java(tm) platform

package org.jfree.chart;

import java.awt.Color;

/**
 * Class to extend the number of Colors available to the charts. This
 * extends the java.awt.Color object and extends the number of final
 * Colors publically accessible.
 */
public class ChartColor extends Color {

    /** A very dark red color. */
    public static final Color VERY_DARK_RED = new Color(0x80, 0x00, 0x00);
```

Figure 6.3: JFreechart visualization with X-Ray

the code of the project. By exploring the city is possible to find classes with high number of methods or attributes that may indicate a certain level of complexity associated.

6.1.3 Comparison

In order to compare theses plugins, key aspects must be defined. The definition of these aspects was done by considering the visualization basis explained in chapter 2 as well as the experience of the author when using the Eclipse IDE[2]. Therefore, the following five aspects were established:

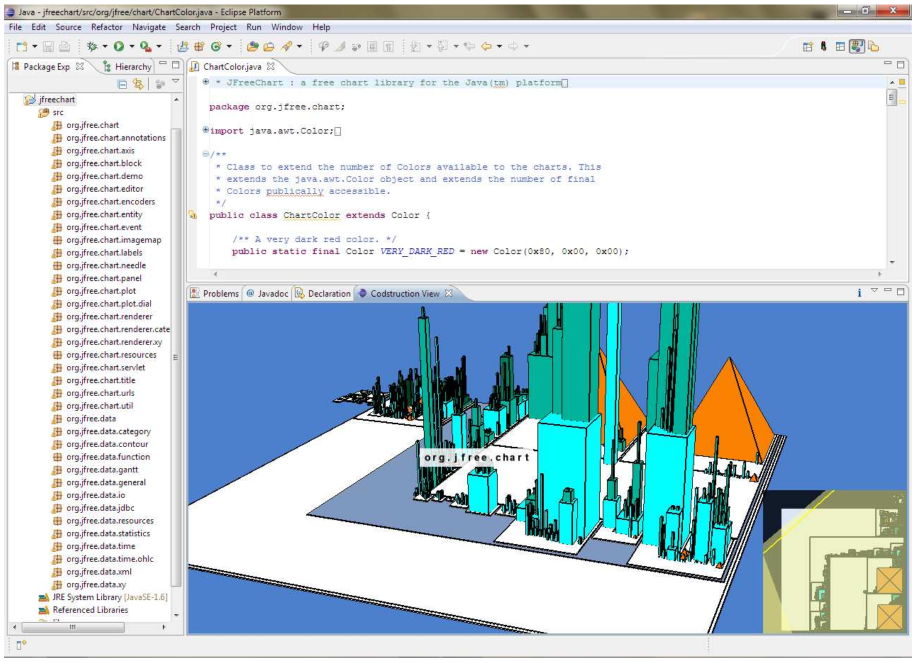
- Visualized information: This item helps to determine the quality of the information and how it will help the user to understand the analyzed system better. Specifically it answers the question *where in the IDE does the information come from?*.

- Type of visualized information: This aspect evaluates if the plugin correctly handles the available information in the IDE. It should answer the question *how the analyzed information is represented by the plugin?*.

- Navigation: As it was seen in chapter 2 when explaining the characteristics of a metaphor, the navigation needs to be evaluated according to the method used by the plugin. This way, this aspect answers the question *how the user is able to navigate through the visualization?*.

- Coupling between source code and views: Coupled views in the IDE allow to easily navigate through the information available within the IDE. In this case the navigation is not performed in the metaphor itself but through the analyzed information. This item responds to the question *is there a link between the views and the analyzed information?*.

[2]Four years using the IDE



```java
 * JFreeChart : a free chart library for the Java(tm) platform

package org.jfree.chart;

import java.awt.Color;

/**
 * Class to extend the number of Colors available to the charts. This
 * extends the java.awt.Color object and extends the number of final
 * Colors publically accessible.
 */
public class ChartColor extends Color {

    /** A very dark red color. */
    public static final Color VERY_DARK_RED = new Color(0x80, 0x00, 0x00);
```

Figure 6.4: JFreechart visualization with Codstruction

- Available metrics: the amount and type of available metrics.
- Keyboard based interactions: This is an important aspect, since within Eclipse and other similar IDEs most interactions can be done via keyboard shortcuts to speed up production. So it should answer the question *does the plugin allow interaction with the application using the keyboard?*.

Table 6.1 shows the aspects above mentioned and how they are faced by each of the three plugins. It highlights the aspects in which SeeIT 3D is better than the others with green color. From this table can be concluded that SeeIT 3D is tightly coupled with the IDE, meaning that it takes advantage of the characteristics of the IDE itself such as, source code model and the coupling with the different views available. It is also important to notice that SeeIT 3D employs a set of well known metrics that perform syntactical as well as semantical analysis on the source code, which leads to a very complete view of the system that allows to understand and maintain the system easier than browsing the source code. Besides, SeeIT 3D not only visualizes source code, it can also visualize different sources of information by using the XML visualization that the other plugins do not have. Finally in this table can be seen that SeeIT 3D is the only plugin able to handle user interactions via keyboard. This method of interaction is widely used by developers using this IDE on their daily work.

6.2 Evaluation Framework

SeeIT 3D is aimed to support software visualization as a way to understand software systems. These types of visualization systems are usually evaluated within some kind of environment to validate their functionality and usefulness, and SeeIT 3D is not the exception. This environment is provided by an evaluation framework. There has been proposed different evaluation frameworks, most of them rooted in the work done by Price *et al.* [34].

Aspect to consider	SeeIT 3D	X-Ray	Codstruction
Visualized Information	Source code but can be extended using the XML based visualization	Source code	Source code
Type of information visualized	Metric based information. It allows to visualize relationships using different metaphors	Metric based information. It visualizes relationships by defining an extra view	Metric based information. It visualizes hierarchical relationships
Navigation	Basic 3D navigation commands (rotate, translate and pan)	Scrolling when view is too big for the screen	Basic 3D navigation commands (rotate, translate and pan). It also offers a mini-map to navigate easily
Coupling between source code and views	Based on a link between view and package explorer. It also offers open editor when double click on a visualization element	Based on a link between view and package explorer	Open editor when double click on visualization element
Available metrics	Four built-in metrics: LOC, Complexity, Lack of Cohesion and Control structure[3]	Three metrics available: LOC, Number of methods and Class type	Two metrics available: Number of methods and Number of attributes
Keyboard based interactions	A high number of interactions can be performed using the keyboard	None	None

Table 6.1: SeeIT 3D Vs. X-Ray Vs. Codstruction

With the intend to evaluate the plugin with one of these frameworks, three of the most important were considered: Price *et al.* [34] in 1992, Gallagher *et al.* [10] in 2008 and the work presented by Storey *et al.* [39] in 2005. The first one was discarded as the other two rely on this work to make it better and more consistent with the new technologies and trends. The work from Gallagher is focused on software architecture visualization which makes it not fully suitable with SeeIT 3D, while the work proposed by Storey *et al.* is more focused on the software development aspect. These characteristics lead to choose it to perform the evaluation of SeeIT 3D.

6.2.1 Framework Description

Although the dimensions proposed by Storey cover almost every aspect of software development, there is always room for improvement as it was shown in [32], where an extended version of the Storey's framework was presented. Considering this extended framework, the complete set of dimensions used for the evaluation of SeeIT 3D is described below:

1. *Intend*: It defines the main goal of the tool and the reasons that lead to its design and implementation. Within this category, the end user identification is performed. For instance, architects, software analysts, developers, re-engineers, etc. Also, the time is mentioned; a concept that refers to the temporal range of the analyzed data, which can be present time, recent past or evolutionary history. Finally, the cognitive support is added, which is used to capture the tool functionality that allows users a better understanding of the analyzed system.

2. *Information*: It groups the data source used by the tool to generate the analysis and visualization. This dimension defines categories of information such as: source code, change administration, bug tracking, documentation and formal communication.

3. *Presentation*: It is the mechanism used by the tool to show information to the user. Aspects like form allow to identify how suitable are the visual resources used. This dimension makes a differentiation of the types of views offered by the visualization tool; they include dependencies graph, statistical charts as well as structural complexity graphics. The use of novel presentation techniques will always be an added value, such as the visual variables, granularity level and animations.

4. *Interaction*: It refers to the interactivity level offered by the tool. This dimension includes batch/live analysis, customization and personalization settings, query mechanisms and the level of navigation. In online mode the user gets the information at the same time he manipulates the view. In batch mode, external scripting programming is necessary. On the other hand the customization level is evaluated as well as the query mechanisms. Finally the ease of navigation is evaluated, where aspects such as orientation and placement are analyzed.

5. *Infrastructure*: It refers to the software or hardware requirements needed to run the visualization tool.

6. *Support*: It evaluates the availability of help resources as well as explanation about the functionality of the tool. It includes forums, web pages, tutorials and videos that allow the user to understand the visualization system easily.

6.2.2 Evaluation

The next subsections present the result of applying the described framework to SeeIT 3D.

6.2.2.1 Intend

As mentioned through this document, the main aim of SeeIT 3D is to provide three dimensional views of a software system that allow developers to understand its components and interactions, identify problems or bad smells, and analyze some of its properties. According to the used framework, this dimension can be studied through subcategories:

- Role: The developer is the main user of an IDE, so he is the main role involved in SeeIT 3D. Other stakeholders are able to take advantage of the visualization such as software architects (to visualize information less granular than a developer would do) and re-engineers (to support tests or redesign tasks).
- Cognitive support: As it was explained in chapter 3 the employed metaphor exposes a view that can be changed as user wants. This view includes a set of containers that represent previously analyzed artifacts. Each of these containers is composed by polycylinders that represent artifacts in a lower granularity level that the container; these polycylinders have three visual properties: height, width and color. The view also allows the user to determine what metric value a polycylinder has by mapping it to a certain visual property. However the view is not limited to always represent a metric with the same visual property. Sometimes it is easier to determine metric values by applying it to a different visual property. This way, SeeIT 3D is able to handle different software metrics mapped to different visual properties depending on the user's needs, so it is possible to observe a system from different points of view. Besides of the visualization of containers/polycylinders, the view also allows to observe the relationships between software artifacts. These relationships are represented using different mechanisms as stated in section 3.1.1. Therefore a container is said to have a relationship with other if there is a certain visual property applied to it, for example a line that connects it with other. Using all this visualization support is possible for the user to navigate through the system, using the view, which allows him to understand how the system is built as well as helping him to find possible code smells or design problems.
- Time: SeeIT 3D analyzes only one version of the system, so it is not suitable to compare or study sequences of versions to get evolutionary data. However, by using the XML based visualization the tool can be fed with information related to other states or releases.

6.2.2.2 Information

The framework establishes the following categories of sources of information:

- Change management: In this case SeeIT 3D does not offer change management; however the XML based visualization can be used as stated previously.

- Program code: This is the main source of information for the tool. It performs syntactical analysis such as the number of lines of code and semantic analysis such as McCabe complexity and Lack of Cohesion. This kind of information provides high amounts of data about the software system.
- Documentation: SeeIT 3D does not take information from the source code documentation, requirements, testing or design diagrams. Although once again, it can be used an XML file to give information to SeeIT 3D in order to render it within Eclipse.
- Other sources: The current version of SeeIT 3D does not support additional sources of information within its *modeler component*; however it offers XML based visualization for other sources of information.

6.2.2.3 Presentation

This item differs from tool to tool because each one applies its own metaphor. So its own mechanism to present information is defined. Nonetheless, the framework allows us to consider some key properties such as:

- Form: SeeIT 3D uses a graphical method based on a three dimensional world to present information to the user. Using a third spatial dimension is possible to render more information than using a 2D space.
- Types of views: The plugin offers a three dimensional view composed by polycylinders, containers and relationships representations. In some cases this kind of relationships can be seen as a dependency graph if the relationships are represented by lines or arcs. Each of these elements represents the value of a certain metric of the viewed artifact. The different views of the same artifact (when changing the mapping and granularity level) can be seen as different types of views, as they offer a completely different point of view of the system.
- Techniques: SeeIT 3D makes use of several visual variables like color (and different color scales), position (to mark containment relationships), sizes (to express the value of a metric) and geometrical shapes (to show relationships). Also the software artifacts can be seen at different granularity levels, as well as a small type of animations when viewing relationships with this mechanism.

6.2.2.4 Interaction

The framework indicates that the interaction characteristics include the following:

- Online/batch: The method used by SeeIT 3D is online because the main source information changes constantly. When viewing source code the batch method is not appropriated because it would be a time consuming task.
- Personalization: SeeIT 3D offers several customizable properties as it was explained in section 4.1. Besides these properties, the metaphor itself can be customized. Changing the mapping between visual properties and metrics is a type of customization as well as when changing color scale or relationships generator.

- Navigation/Orientation: The plugin offers a set of general navigation commands within the visualization area (zoom, rotate and translate). Other kinds of navigation included in the plugin are: view to code editor, editor to view, view to package explorer and vice versa. View to package explorer navigation is considered as a coupled view because the package explorer is a visualization of the system, although it is not include by the plugin itself.

6.2.2.5 Infrastructure

A couple of infrastructure items need to be considered:

- Operating system: Can be one of Windows, Linux or Mac OS. This flexibility is provided by the IDE itself, as well as the Graphics engine used.
- Eclipse version: Eclipse 3.5 (Galileo) or newer.

6.2.2.6 Support

The aspects to consider, as specified by the framework, are:

- Web page: The plugin has a web page located at: http://code.google.com/p/seeit3d/
- Tutorial: It does not include tutorial.
- Videos: A general overview of the GUI and how it can be used.
- Mailing: It does not have mailing list.
- Forum: It does not have forum but a Wiki system offered by Google Code.

6.2.3 Evaluation Results

The framework highlighted the most important aspects of the tool. It allowed to demonstrate that the main goal of SeeIT 3D is to help developers (and other stakeholders) to understand and comprehend a software system as a whole. This is achieved by using a flexible and generic metaphor in which is possible to navigate through the analyzed system in order to understand its characteristics. This metaphor is flexible enough to visualize data not only from a specific version of the source code but from several sources information such as: recent or past history, documentation, runtime and practically any other source whose analysis can be expressed using the XML format specified by the tool.

It was also possible to find that the plugin employs a 3D space which allows to visualize more information that in a bidimensional world. SeeIT 3D is able to use this 3D space to extend metaphors used in 2D, for example when displaying relationships using the lines/arcs based representation, this way it is able to render the same information as a tool using 2D. However the visualization is not limited to the extension of 2D metaphors, it offers different views of the same artifact when the mapping between visual properties and software metrics is changed; even it has coupled views when using the link between package explorer and the SeeIT 3D view.

The evaluation of the representation dimension highlighted that the plugin uses sizes of geometrical shapes, position of the elements as well as a wide spectrum of colors represent information. However it also showed that extra elements could be used, for example different geometrical shapes to display more elements associated to the analysis of the system.

With the evaluation framework it was also shown that SeeIT 3D employs an online visualization mechanism which simplifies the visualization process by keeping an up to date information about the software system. Besides, it was seen that the visualization offers a set of common manipulation tools for a 3D space as well as several customizable options, so it is possible for the user to change the appearance as he wants.

From the infrastructure point of view, the framework showed that SeeIT 3D has flexible requirements about it. It can be run in different operating systems due to the flexibility of the IDE.

Finally the framework allowed observing that there is a lack of documentation and communication mechanisms between tool developer and users of it. This is caused because SeeIT 3D is an experimental tool whose development is at early stages, besides it only has one developer.

6.3 Summary

This chapter presented how SeeIT 3D was evaluated. First, the plugin was compared to other similar tools embedded within the Eclipse IDE. This comparison allowed to see that SeeIT 3D performs well in this kind of environment because it makes use of the adequate controls for each task. Next, an evaluation framework was chosen and the tool was evaluated against it, showing that SeeIT 3D fulfills a wide spectrum of developers needs.

Chapter 7

Conclusions and Future work

This document presented how SeeIT 3D was designed and tested as a software visualization tool. It also included analysis about metaphors, IDEs, similar visualization tools and evaluation methods of these types of systems. They consider information that can be gathered from three different sources of information: *static*, *dynamic* and *evolutionary* information. *Static* refers to the data that can be extracted from a environment that does not need the program to be running, *dynamic* indicates that the program needs to be running in order to extract information and *evolutionary* information refers to how the program and its properties have evolved through time. After the analysis is completed, the visualization itself is carried on. This process is performed by loading the analyzed data into the metaphor, and then the user is able to explore, manipulate and dig into the software system represented by the visualization.

From the concepts explained before, it is defined the basic visualization process: gather information, analyze it and visualize the results. This process can be seen in different software analysis methods like mining software repositories. This characteristic makes the visualization process itself expandable to other areas where a visualization mechanism is needed. This is why SeeIT 3D provides an external input data that allows other software analyzers to feed the metaphor with generic information.

In order to build SeeIT 3D, there were considered several aspects which included: the metaphor, the metrics, the IDE, the underlying technologies and the user interaction. The metaphor developed is based on the sv3D metaphor; it allows displaying data gathered from different sources of information. Besides, it provides a powerful and simple mechanism that, when it is used together with metrics such as the Lack of Cohesion, allows to visually detect emerging patterns in the software. It leads to an easy understanding of a software system and its characteristics.

On the other hand, the extensibility and popularity of the Eclipse IDE has brought additional tools to software development. This popularity and acceptance made it become the best option to build a visualization tool like SeeIT 3D. Together with technologies like Java 3D, this IDE allows to provide a useful environment where the developer can easily interact without needing to propagate the changes between two tools as in the case of other visualization systems.

The user interaction in a software visualization tool is very important aspect that needs to be considered. In SeeIT 3D the GUI was developed considering usability principles such as: commonly used controls and simplicity of the exposed tools. These principles produced the view explained before; it allows to easily bind a metric to a visual property, changing from one view to another making the manipulation faster and more useful for the user.

The visualization provided by SeeIT 3D demonstrated that is possible to understand a software system from a different perspective, not only from the source code but from a visual point of view. The different exposed views, allowed comparing how different systems have their own characteristics according to the task they perform. Although the functionality of SeeIT 3D was tested by a set of real world projects, it needed to be evaluated with an evaluation framework that made possible to define the dimensions where it behaved better or worse than other visualization systems. This evaluation highlighted that SeeIT 3D is a powerful tool that allows understanding software systems by providing a flexible metaphor that the user is able to customize as he wants. However, it showed that the tool lacks of a well defined documentation and training mechanisms, because its experimental approach.

Although SeeIT 3D provides a platform for viewing software, there is future work to do; it includes providing better 3D manipulation tools like the ones included in CAD systems that make the manipulation more simple and intuitive. Also, in this respect is necessary to complement the view with more attractive components such as better lights management, reflective objects and better spatial representation that allows to easily understand the three dimensional world. A second aspect to consider in future work, is related to provide an *extension point* within the Eclipse platform that allows other plugins to easily use the visualization framework exposed by SeeIT 3D; it would be complementary to the XML based visualization but with the advantage of making the visualization process more simple because it would be done without any intermediate steps. Finally, probably the most important task to do in the future refers to the evaluation process with a group of developers whose feedback can be used to improve the tool based on real user's needs.

SeeIT 3D achieved the goals proposed at the beginning of this work, it provides a three dimensional view in which is possible to visualize relationships and metrics associated to them; it is included within an IDE where the source code is maintained; it provides a XML based visualization that enable other tools and sources of information to feed the visualization; it allows to easily change the mapping between software metrics and visual properties and it has four useful metrics to highlight software characteristics. This way, SeeIT 3D is a step forward in the development of software visualization tools and can be improved by others in order to continue growing and making software visualization a better mechanism to understand and maintain systems.

Appendix A

Information about the SeeIT 3D Development

A.1 Research Production

During the development of SeeIT 3D there were presented two works:

1. Montaño, D.; Aponte, J.; Marcus, A.; , "Sv3D meets Eclipse", *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International/Workshop on*, vol., no., pp.51-54, 25-26 Sept. 2009.

2. Montaño, D; Aponte, J.; , "SeeIT 3D: un plugin de Eclipse para visualizar y comprender código fuente", *Conferencia Latinoamericana de Informática.* 18-22 Oct. 2010.

A.2 Additional Resources

In https://code.google.com/p/seeit3d/ is located the web page of the project. There is possible to find:

- The user manual with information about installation process and how to start visualizing projects
- A bug tracker system
- A wiki system with information about the releases of the tool
- The source code of the tool
- The update site of the plugin. It is located at http://seeit3d.googlecode.com/svn/trunk/SeeIT3D_Update
- The XML schema definition for the XML based visualization

Appendix B

XML Example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<seeit3d:containers
      xmlns:seeit3d="http://seeit3d.googlecode.com/files/seeit3d.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://seeit3d.googlecode.com/files/seeit3d.xsd">
   <seeit3d:container name="idContainer1" granularityLevelName="just this" related="
        idContainer2 idContainer3" visible="true">
      <seeit3d:metricsList>
         <seeit3d:metricDescription type="CONTINUOUS" max="25">ABC</seeit3d:
             metricDescription>
         <seeit3d:metricDescription type="CATEGORIZED" categories="text 123 ODF">DEF</
             seeit3d:metricDescription>
      </seeit3d:metricsList>
      <seeit3d:mapping>
         <seeit3d:mappingValue>
            <seeit3d:metricName>ABC</seeit3d:metricName>
            <visualProperty>HEIGHT</visualProperty>
         </seeit3d:mappingValue>
         <seeit3d:mappingValue>
            <seeit3d:metricName>DEF</seeit3d:metricName>
            <visualProperty>COLOR</visualProperty>
         </seeit3d:mappingValue>
      </seeit3d:mapping>
      <seeit3d:polycylinder>
         <seeit3d:name>Poly 1</seeit3d:name>
         <seeit3d:metricsValue>
            <seeit3d:entryMetricValue>
               <seeit3d:metricName>ABC</seeit3d:metricName>
               <seeit3d:value>1.2</seeit3d:value>
            </seeit3d:entryMetricValue>
            <seeit3d:entryMetricValue>
               <seeit3d:metricName>DEF</seeit3d:metricName>
               <seeit3d:value>text</seeit3d:value>
            </seeit3d:entryMetricValue>
         </seeit3d:metricsValue>
      </seeit3d:polycylinder>
      <seeit3d:polycylinder>
      <seeit3d:name>Poly 2</seeit3d:name>
      <seeit3d:metricsValue>
        <seeit3d:entryMetricValue>
          <seeit3d:metricName>ABC</seeit3d:metricName>
          <seeit3d:value>5</seeit3d:value>
        </seeit3d:entryMetricValue>
        <seeit3d:entryMetricValue>
          <seeit3d:metricName>DEF</seeit3d:metricName>
          <seeit3d:value>123</seeit3d:value>
        </seeit3d:entryMetricValue>
      </seeit3d:metricsValue>
    </seeit3d:polycylinder>
    <seeit3d:polycylinder>
      <seeit3d:name>Poly 3</seeit3d:name>
      <seeit3d:metricsValue>
        <seeit3d:entryMetricValue>
          <seeit3d:metricName>ABC</seeit3d:metricName>
          <seeit3d:value>25</seeit3d:value>
        </seeit3d:entryMetricValue>
        <seeit3d:entryMetricValue>
          <seeit3d:metricName>DEF</seeit3d:metricName>
          <seeit3d:value>ODF</seeit3d:value>
        </seeit3d:entryMetricValue>
      </seeit3d:metricsValue>
    </seeit3d:polycylinder>
  </seeit3d:container>
  .
  .
  .
</seeit3d:containers>
```

Bibliography

[1] R. Baecker and A. Marcus. *Software Visualization*, chapter Printing and Publishing C Programs, pages 45–61. MIT Press, 1998.

[2] R. Baecker and D. Sherman. Sorting out sorting. Video shown at SIGGRAPH-81, August 1981. Video.

[3] M. Burch, S. Diehl, and P. Weigerber. Eposee: A tool for visualizing software evolution. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2005.

[4] Rikk Carey and Gavin Bell. *The annotated VRML 2.0 reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1997.

[5] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. The cave: audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–72, 1992.

[6] Andrew Davison. *Pro Java 6 3D Game Development: Java 3D, JOGL, JInput and JOAL APIs*. Apress, April 2007.

[7] S. Diehl. Software visualization. In *Proceedings of the 27th international conference on Software engineering*, pages 718–719, 2005.

[8] S. Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Verlag, 2007.

[9] S. C. Eick, J. L. Steffen, and E. E. Sumner Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18 Issue 11:957–968, 1992.

[10] K. Gallagher, A. Hatch, and M. Munro. Software architecture visualization: An evaluation framework and its application. *IEEE Transactions on Software Engineering*, 34(2):260–270, 2008.

[11] H. H Goldstine and J. Von Neumann. *Planning and coding of problems for an electronic computing instrument*. Institute for Advanced Study, 1947.

[12] D. GraÃ§anin, K. MatkoviÃ§, and M. Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, 2005.

[13] B. Henderson-Sellers, L. L. Constantine, and I. M Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158, 1996.

[14] T. D Hendrix and S. Maghsoodloo. The control structure diagram: An overview and initialevaluation. *Empirical Software Engineering*, 3(2):131–158, 1998.

[15] Pourang Irani, Maureen Tingley, and Colin Ware. Using perceptual syntax to enhance semantic content in diagrams. *IEEE Computer Graphics and Applications*, 21(5):76–85, 2001.

[16] Pourang Irani and Colin Ware. Diagrams based on structural object perception. In *Proceedings of the working conference on Advanced visual interfaces*, pages 61–67, Palermo, Italy, 2000. ACM.

[17] M. A. Jackson. *Principles of Program Design*. Academic Press, Inc., Orlando, FL, USA, 1975.

[18] J. A Jones, M. J Harrold, and J. T Stasko. Visualization for fault localization. In *Proceedings of ICSE 2001 Workshop on Software Visualization*, pages 71–75, Toronto, Ontario, Canada, 2001.

[19] C. Knight and M. Munro. Comprehension with in virtual environment visualisations. In *Program Comprehension, 1999. Proceedings. Seventh International Workshopon*, pages 4–11, 1999.

[20] G. Lakoff and M. Johnson. *Metaphors we live by*. Chicago London, 1980.

[21] M. Lanza. Codecrawler-lessons learned in building a software visualization tool. In *CSMR '03 Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, volume 2003, 2003.

[22] Haim Levkowitz and Gabor T. Herman. Color scales for image data. *IEEE Computer Graphics and Applications*, 12(1):72–80, 1992.

[23] W. Lowe and T. Panas. Rapid construction of software comprehension tools. *International Journal of Software Engineering and Knowledge Engineering*, 15(6):995–1025, 2005.

[24] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5:110–141, April 1986.

[25] J. I Maletic, J. Leigh, A. Marcus, and G. Dunlap. Visualizing object oriented software in virtual reality. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 21–13. ACM, 2001.

[26] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. A task oriented view of software visualization. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 32–, Washington, DC, USA, 2002. IEEE Computer Society.

[27] Jacopo. Malnati. X-ray: An eclipse plug-in for software visualization. Master's thesis, UniversitÃ¡ della Svizzera italiana, 2007.

[28] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis '03, pages 27–ff, New York, NY, USA, 2003. ACM.

[29] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32:1415–1425, December 1989.

[30] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22, 2005.

[31] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8:12–26, August 1973.

[32] Y. Nino, J. Aponte, D. Montano, and V. Collazos. Aplicacion de un framework de evaluacion a plugins de eclipse para visualizacion de software. In *IV Congreso Colombiano de Computacion*, 2008.

[33] OMG. Uml specification. http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/.

[34] B.A. Price, I.S. Small, and R.M. Baecker. A taxonomy of software visualization. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume ii, pages 597 –606 vol.2, January 1992.

[35] Joel T. Stasko. Using student-built algorithm animations as learning aids. *SIGCSE Bulletin.*, 29:25–29, March 1997.

[36] John T. Stasko. Tango: A framework and system for algorithm animation. *ACM SIGCHI Bulletin*, 21:59–60, January 1990.

[37] J.T. Stasko and J.F. Wehrli. Three-dimensional computation visualization. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 100 –107, August 1993.

[38] Margaret-Anne Storey, Casey Best, and Jeff Michaud. Shrimp views: An interactive environment for exploring java programs. *International Conference on Program Comprehension*, 0:0111, 2001.

[39] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 193–202, New York, NY, USA, 2005. ACM.

[40] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. Cvsscan: visualization of code evolution. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 47–56, New York, NY, USA, 2005. ACM.

[41] Colin Ware, David Hui, and Glenn Franck. Visualizing object oriented software in three dimensions. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, CASCON '93, pages 612–620. IBM Press, 1993.

[42] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, pages 921–922, New York, NY, USA, 2008. ACM.

[43] Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. Visualization of cvs repository information. In *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on*, pages 231 –242, oct 2006.

[44] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.