Växjö University

School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

Visualization of Text Duplicates in Documents

Chao Wang & Han Pan

June
2009

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 09029
ISSN 1650-2647
ISRN VXU/MSI/DA/E/--09029/--SE

Master Thesis

Visualization of Text Duplicates in Documents

Han Pan & Chao Wang

2009

Department of Computer Science
School of Mathematics and Systems Engineering (MSI)
Växjö University

Supervisor:
Prof. Dr. Andreas Kerren

Abstract

In this thesis, a tool to visualize duplicate parts in a series of given documents is developed.

Text duplicates are very common nowadays in all fields. This behavior severely harms the rights of the original authors though it facilitates the work of those who copy from them. Effective legal measures have been taken when it comes to copyright issue. An increasing large number of people have paid serious attention to what they write when they refer to other people's works. Although references are properly made by many who admire and respect others' achievements, plagiarism takes place all the time. Therefore, an intuitive way of visualizing duplicate parts is needed so that people can easily grasp the purpose and decide the legality of those duplicates. When it comes to computer science, software clone is very typical phenomenon among different development groups or even within one group. Since a piece of software usually have its hierarchy, it is also interesting to group members when they do a clone detection of their own or other software. For example, if a good overview of the hierarchies is provided in a tree representation, one can easily locate the clones of a particular node in other trees. More interaction techniques can allow concrete code accesses through double clicking on a highlighted node.

To visualize duplicate parts in a nice and intuitive way, a visualization tool is developed for this thesis project. By the time it is done, the following features should be fulfilled. First, the tool can visualize similar or identical parts given a data set. Second, hierarchies of those files can be demonstrated with proper layout. Third, the user can manipulate the data items on the screen in order to get a better insight of the data set and help with analysis tasks. Forth, different levels of abstraction are provided so that the user can either get an overview of all the files or specifically check the duplicate parts in the documents of interest.

Keywords:
Duplicates, PREFUSE, Visualization, Treemap, Similarity, Interaction.

Contents

List of Figures

1 Introduction

In this chapter, a brief introduction of the thesis will be given. First, what we are going to solve in the thesis will be explained using examples from the real world. Subsequently, goal criteria with which the program stands out other similar ones will be discussed in general. After that, the motivation for developing this kind of programs is conveyed so that the potential users of the program may feel interested.

1.1 Problem Issued

As time goes by, increasing attention has been paid to the matter of copyright. Since any decently written book, document or innovative design of a system is the product of people's hard work, a great number of readers or users, as well as the authors themselves, insist that the copyright of those works should be protected properly by law. However, although positive measures have been taken for this issue, there are always some people who want to take the benefit resulting from others' work without proper allowance. Plagiarism takes place everywhere, especially in the field of academy. When it comes to software development, borrowing code will affect the original author's profit in a very bad way. For instance, certain excellently designed pattern may be duplicated to serve functionality in a new piece of software without permission from the original designer. This is not decent behavior and may trigger legal charges. On the other hand, within the same development group code clone can also take place. In this case, the detection of those similar or identical parts can help the members with maintenance of the duplicated component of a piece of software. Assume that an algorithm has been borrowed in order to fulfill the same functionality for other software. By knowing where the duplicates exist lead to ease of bringing the software back to normal once it fails to fulfill its tasks due to the defects of the borrowed algorithm, provided that the defects have been reported or no such failures have once taken place before the algorithm was integrated. What can we do with the problems mentioned above?

Since plagiarism can by no means be avoided, people who are concerned with intellectual property right have made lots of efforts to find out whether duplication exists in books, papers and documents. For instance, the so-called "anti-cheating machine" has disillusioned countless students who tend to take credit for others by giving out text-based analysis results of the submitted assignments. The result is a percentage along with the corresponding reference to the source from which the machine considered to be copied. This kind of textual results surely play an active role in duplication detection. However, when the size of the data set goes increasingly big, and it is, people will probably get stressed when they try to find the information in which they are interested, not mentioning to gain any insight from the data set. Is there a more effective and efficient way to do the job? At present, information visualization has become an increasingly heated topic. More and more people are devoting their efforts to the advance of visualization technique. A picture says more than a thousand lines of words. Visualization is to create a mental image or mental model of something so that the viewer can get information or an insight from the

original data, which is the major task of information visualization. Therefore, we can say that the problem aforementioned will be solved simply by a series of graphs which are well drawn and organized. They can save lots of efforts to find the duplicates in files and even provide some higher-level abstraction that textual results can never be able to. To the best knowledge of the authors, no such visualization tool which specially deals with text duplication has been put into public use. Therefore, we are doing this thesis in order to provide potential users with promising software that helps with plagiarism detection.

1.2 Goal

The purpose of this thesis is to develop a visualization tool which can solve the problem stated above in the scope of software development.

The main function of the tool is to find text duplicates in a series of given files, such as java source codes, XML documents and the like. Specially, the tool that is to be implemented is supposed to be able to analyze the given files, find identical or similar parts (given a threshold) between them and then display the results in a visual way by utilizing some information visualization techniques.

Furthermore, a program's source code or an XML representation of entities in real life usually has a certain kind of structure or, more precisely, hierarchy. Our visualization tool should have the ability of detecting the relationships among those programming entities and present them in the same way as it does with text duplicates. This provides a higher level of abstraction so that it is easier for the potential user of the tool to decide if plagiarism does happen in the files to compare as well as to locate the approximate place.

More interesting is that different colors will be chosen for the duplicate texts to indicate the extent of similarity. For instance, if two pieces of codes are identical, they will be highlighted with the color of red while non-duplicate parts remain black-font. Intermediate colors will be set to the texts accordingly.

At the completion of this tool, the potential user will be able to identify the similarities of two groups of files on different levels of abstraction and to make decisions about whether the duplicate is serious enough to be called plagiarism.

1.3 Motivation

The importance of this project lies in the following points:

- On behalf of teachers and professors at universities, this tool will help to check if a student has borrowed any code/idea/words (which is very common on campus) from a source without the permission of the author or proper referencing.
- From a business perspective, a project manager can verify the decency of his/her fellows' work and take measures with proofs if the analysis result of our tool brings surprises.
- More critically, the stakeholders of a company can prevent their product being charged of plagiarism by applying analysis on it before it is released for public use. Otherwise, suitcases that affect the company adversely be incurred.

1.4 Report Structure

To get the reader familiar with the field of information visualization as well as the way we develop the software in this case, the chapter followed will explain all the relevant theories, definitions and terminologies. Also, a refined objective will be seen to further the discussion about the procedure itself.

The rest of this thesis will present the process of finding a solution to the problem mentioned. In Chapter 2, related expertise will be introduced so that the reader can understand the techniques we use to solve the problem, including visualization techniques and the process we do the development. Chapter 3 introduces some related works that deals with the similar area as we do and convinces you how our tool can stand out others. Followed is Chapter 4 which explains the visualization approaches we use to realize our tool. Chapter 5 demonstrates how we implement those outstanding approaches and the result will also be shown. Chapter 6 concludes this thesis with the achievements we have made and the future work we should do to improve the implementation.

2 Important Aspects of Information Visualization

In this chapter, related theories helping understand the whole thesis will be explained with examples. Section 2.1 gives an introduction of basic concepts and human perception. Section 2.2 focuses on the reference model, with which we carry out the process of implementation. Section 2.3 acquaints the readers with different representations of data, among which treemap is chosen to be the major one in this project. Overall, this chapter is mainly based on Professor Kerren's lectures for the course "Information Visualization I, Spring 2009, Växjö University".

2.1 Information Visualization and Human Perception

Nowadays, more and more attention is being paid to the world of information visualization. With the invention of computers, researchers who devoted their efforts to this field have successfully visualized large amount of textual data which are hard to interpret with nicely-drawn and intuitive graphs, which even and often lead to better understanding of the original data and insights into them. The extra information revealed after the visualization also helps with making decision on important things, like whether the expense should be cut since the balance is not that balanced according to the graph. So far, the benefit of information visualization sounds very promising. However, before going further into it, it is necessary to have a clear idea of what information visualization is.

Though information visualization has not been brought on the table for long, people started to do researches on this field long before the first computer in the world was invented. Among those researches, there are some that are considered as excellent in the history of information visualization. For example, Minard's map of Napoleon's march to and retreat from Moscow, Nightingale's diagram depicting the number and rate of people's death in the hospitals and Harry Beck's underground map. Let us look into Beck's map further to get a concrete impression of information visualization.

Bill Bryson (1998) [2] said Beck "realized that when you are underground, it does not matter where you are." That's to say, we may totally disregard the situation on the ground when trying to draw a map of all the subway lines. For people who are on the subway, the things that matter are where they are going and with which lines can they go with. Having seen this point, Beck scaled up and down to maintain the elegance of the map while maintaining the correct sequences of stops, as illustrated in Figure 2.1. The result is a brand new London regardless of the real geography in the upper world. The great achievement was not paid enough attention to at the time of Beck's. However, his masterpiece does play an irreplaceable part in drawing transporting maps all over the world. This is a successful visualization of underground system in London, which facilitates people's, especially visitors' life. With this map, tourists who are in London for the first time can probably plan their trips easily [2].

By seeing the figure in the example, it is easy for people to arrive at the conclusion that information visualization is just the production of a graph. Actually, that is just part of the job. Usually, Visualizing data is a human cognitive process during which a mental image or model is generated by the viewer. The purpose is to

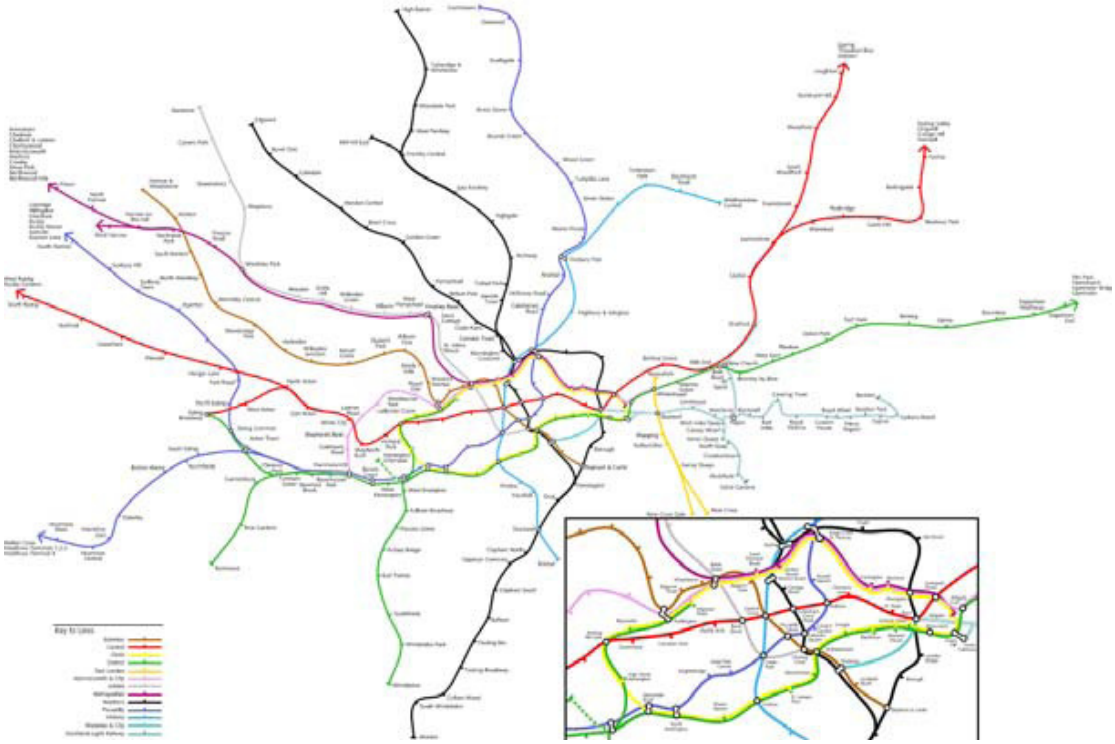

Figure 2.1 Beck's Map of London Underground, taken from [1].

discover something new from the original data, help with decision making with those found or make an explanation of unknown phenomenon. Hence, information visualization is the process in which textual data are transformed in some way into graphical representation, a mental image/model appears in the viewer's mind provided that the visualization is good and then insights into the data are gained.

Since people have to look at the pictures to do their analysis work on the data set, what can be done to make human better interpret the visualized data is of importance.

When it comes to perception, it is necessary to study the process of it, that is, how human see thing. Basically, there are three steps which are always in sequence. The first step is a low-level parallel processing of so-called preattentative features in an image. In this step, features like color, shape, size, texture draw the views' attention immediately. For instance, in Figure 2.2 the viewer can distinguish the red spot from the blue ones right after they see it. The respond time is within 60 milliseconds [3].

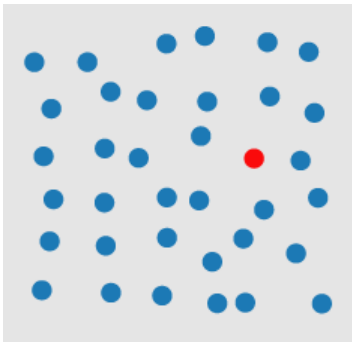

Figure 2.2 Preattentative Feature – Color, taken from [18]

What we can learn from this point is that highlighting critical parts in the scene will probably help the user's interpretation of the visualized data. In our project, identical parts in source files are usually of the viewer's interest. Therefore, encoding those duplicates in hierarchical level in an effective and efficient way is one of the demanded requirements for our visualization tool. In the contrary, if duplicates are vaguely displayed on the screen the viewer may find it hard to get what he wants so that it takes more time to identify them in the scene and finally the efficiency is decreased. Concrete ways to highlight important elements will be demonstrated later in this thesis.

The speed of processing information is relatively slow on the second level of human's perception. During this period of time, patterns are recognized in a sequential manner instead of in parallel as it is in the first step. In this step, people will be aware of complex patterns, contours, regions and so on. Working and long-term memory start to take part in the cognitive activity which means the viewer begins to think about the scene. At the moment, he pays more attention to "arbitrary" symbols which are harder to understand than preattentative features, easy to forget (that is why memory is working), expresses more than those in the first step and may change with time. For example, if a complex formula in physics appears in the visualization, few people would understand what is being conveyed because they lack the expertise to comprehend all the terms which seem Martian. In our case, although the users are probably computer science people it is still necessary to simplify the symbol representing a programming element in a source file when the hierarchical overview is given, such as a rectangle with a label that describes the demanded attributes of the element according to the user's wish.

After those two steps, the viewer has already got a relatively sufficient understanding of what he is looking at. Now, he probably starts to question about the things in which he is interested or seek for data items that he knows may help with his analysis tasks. At this moment, the viewer is already in the third phase of perception that is a sequential, target-oriented processing [3]. In this phase, people will have a clear aim in their minds, for instance "where is the subway station which is nearest to the 'Big Ben'?" Therefore, he does the search along the different lines which stop at the Big Ben. This is obvious a sequential process because normally no one can look at two lines and mentally travels with both at the same time. As for this thesis project, the user of the visualization tool is supposed to check the concrete code in the source files. It happens when he has already found those highlighted parts in the hierarchical overview and wants to go further in order to see whether it constitute plagiarism. At this point, the viewer may not just look at the code, but discuss it with his colleagues and points out the identical lines as well with the help of his speech and motor system, which is also part of the third phase of how human perceive things [3]. The process mentioned above can be demonstrated with Figure 2.3:

So far, the general concept of information visualization has been introduced, which is the process of re-presenting raw data, which are usually in the form of text, with vivid graphical elements in order to acquire a better understanding so that people

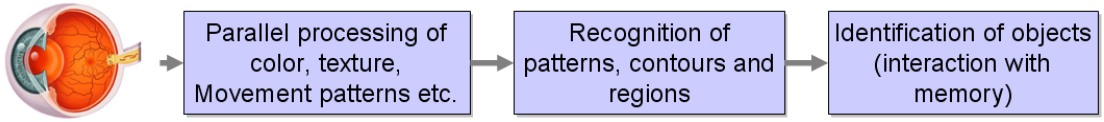

Figure 2.3 Human Perception Process Model, taken from [3].

can draw conclusions upon them. However, in which way can textual data be transformed into graphs which are of totally different representation? In our case, how can a piece of code be converted into visual stuff and still make sense to the user? The following sub-section will talk about the information visualization reference model which is the usual process of solving the problem above.

2.2 Information Visualization Reference Model

Usually, the data which we are about to visualize are of any textual form. They can be a whole paragraph of text in a document describing some statistical stuff from human resource department about the personnel transfer during a certain period of time. It is also possible that news from all over the world is assembled together to do some kind of research so that all the articles are needed to be represented in a graphical way in order to be utilized in the analysis. Raw data like this are very difficult to be converted into graphical elements so that they make sense. Therefore, the first thing we do to visualize them has nothing to do with graphs or any kind of visual view.

To make raw data convertible, data transformation is necessary during which randomly scattered data are re-organized into a more structured form. Only in this way can the data be connected to visual elements and their attributes be depicted by both graphical primitives and texts. In which way can data be re-structured? The conventional way is to use data tables. When put into data tables, for example in a database, underlying relations between different kinds of data become obvious or at least it is less effort-consuming to go through those data before the transformation.

	Understandability	Learnability	Operability	Attactiveness
	1.603448276	1.103448276	1.103448276	6.563218391
LOC	-0.022988506	-0.011494253	-0.011494253	1.586206897
WMC	-0.034482759	-0.034482759	-0.034482759	1.701149425
DIT	-0.034482759	-0.034482759	-0.034482759	1.045977011
NOC	1.793103448	1.793103448	1.793103448	1.793103448
DAC	-0.017241379	-0.017241379	-0.017241379	-0.034482759
TCC	1.022988506	0.511494253	0.511494253	1.022988506
LOD	-1.103448276	-1.103448276	-1.103448276	-0.551724138

Figure 2.4 An Exemplary Data Table.

As it can be seen from Figure 2.4, this data table which describes the metric values of a part of a graph library along with their contributions to 4 aspects of quality is of 2 dimensions. Usually, a data table contains the following things. The first one is the data item, which in the case of the table above, is the library itself. When there are more than just one data items or cases, they will be explicitly shown in the table so

that visualization people can speculate about the amount of work they are supposed to accomplish. The second component is the set of attributes whose values, which belongs to the third group of elements, inform the conditions of the data items. The attributes can be categorized into three types. Nominal attributes are usually interpreted as the names of the data item while ordinal ones are used when the order of the values matters, like the year of production. A third category of data table attribute is very common, that is, the quantitative values, which means the calculation (addition and so on) of those values make sense to the table's user. For instance, later we will present a possible way of assigning similarity values to each programming element displayed in our visualization screen. Since each Java source file has some kind of hierarchy, the similarity of a class to another one, may be worked out through adding all similarity values of its methods' multiplied by a certain weight. It shows that the similarity values of all the programming elements in the original data set are quantitative.

During data transformation, problems can arise when people accidentally write a value wrong. Apart from that, data are possibly missing or their formats and types are changed because careless job or some poorly-implemented automatic converter. Those are what we should pay attention to during this phase.

Data transformation is like a warming-up process which re-organizes raw data in such a way that visualizing them will become easier. The real critical process of information visualization is called visual mapping, which is using graphics technique to re-present the context of data tables so that the viewer can get a vivid impression of the raw data. In reality, searching for a proper visual mapping is absolutely not an easy job although the field of information visualization is already advanced during the last 15 years. Since visualization has a close connection with its application area, potential clients from various fields will probably impose their unique requirements on the visualization tool in order to make it suitable for their jobs respectively. Therefore, there is no convention regarding "best" visual mapping at present. However, excellent practices do exist when it comes to a certain kind of data sets. Although there has not been any standard, two criteria can help us decide whether a certain way of visualization is good. The first is *expressiveness*. A data table can contain lots of information in it and all the valid data are supposed to be visualized in this application. Any loss of originals may cause inaccuracy when the user tries to draw conclusions or make decisions from the incomplete data set. Therefore, the criterion expressiveness requires the integrity of all the data, namely each data item as well as its corresponding attributes should be demonstrated in some way on the screen. Also, only those data in the table(s) should appear because additional information may be generated in order to make visual mapping feasible or easier. This kind of information can happen to be distractive so that the viewer takes it into consideration when doing his analysis tasks. For those who are familiar with the original data set, the extra information will not take any effect. Unfortunately, not all of the viewers are this professional or do not have a chance to see the data tables. In this case, inappropriate insight may be found, which means the visualization is misleading while it should provide more straightforward views. What's more, a good

visualization is supposed to be *effective*. The aim of information visualization is to offer a better understanding of the raw data in any form. It should make it less effort-consuming interpret the original information. Therefore, an application of information visualization can be considered as good if the differences between the data items become clearer and more obvious or those data items which seemed to make a person error-prone are now well-organized and do not have that quality any more. Effectiveness is important in visual mapping. If handled in the wrong way, the resultant visualization may be less interpretable than before or even make no sense at all.

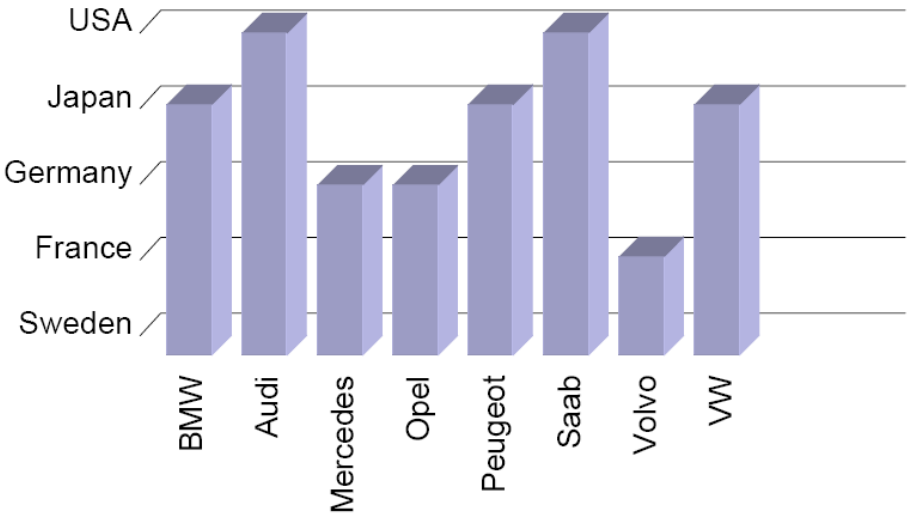

Figure 2.5 Sample Visualization, taken from [4].

Looking at Figure 2.5, one cannot tell whether it depicts the sales of cars of those brands below in each country on the left of this image or it is the production that the original data holder wants to demonstrate here. Apparently, the sample above is not expressive or at least not expressive enough to be a good example of visualization. Perhaps, part of the data, which can be a brief introduction, is lost. In our visualization, hierarchies of source codes are supposed to be laid out. Hence all the programming elements, either it represents a method or a whole file, should be demonstrated in such a way that it is which one is subordinate to another. No single statement is left behind or extra elements are displayed given a level of aggregation (which will be explained in the coming chapter).

There are many ways of mapping data tables to visual structures. However, before looking further into the techniques, another important part of the reference model, which can be considered as the essence of information visualization, needs to be mentioned. View changing is the most interesting part when creating the visualization application and is the “interface” between the user and the computer. Whether a piece of software is successful or not is largely up to how well the view transformation is done. Many designers and implementers make lots of efforts to achieve smooth transformation from one view to another so that the user will not feel any abrupt change from the previous impression. It is precisely because of this part that information visualization becomes an interesting, interactive and worth-exploring

job which arouses increasing concern in recent years.

Although a great number of ways have been applied to various groups of clients in order to make the view to the extent of fanciful, there are generally three types of approaches when it comes to view transformation. Different kinds of *Location Probes* are devised to serve plenty of view control mechanisms. Pop-up windows may be the most common one that allow the user to see more details about the data item in which he is interested. For example, suppose all the pieces of music throughout the 20^{th} century are plotted in a 2D coordinate whose axes represent year and type and each one of them is represented by a recognizable dot. In order to pick out a piece belonging to a certain type from a specific year in order to know the composer, the user will click on the point that represents it. In this case, if pop-up windows are integrated into this visualization the user will get to see the name of the writer as well as other information about this piece of music in a small window upon the original coordinate. This is just a simple application of view changing techniques. Although pop-up windows have defects like they may stop the user checking other items that are near to the focus point, they do help with acquiring more information through interactive selection of the points by the viewer. Similar applications include movable magic lenses under which the data item is displayed in another way or more information will be available. Also, brushing is very useful when the data item exists at more than just one place and once the focused item is selected in a certain way, all the other objects that represent the same item will be highlighted at their own spots respectively. In our thesis project, more advanced techniques will be utilized in such a way that it can ease the job of the view most.

Another category of view transformation technique is called *Viewpoint Control*. As mentioned above, no matter how location probes are used, they do not change the place from where the viewer looks at the scene. However, with viewpoint control, as its name suggests, the position or angel of the viewpoint will be somehow altered. It is not that this technique requires the viewer to move his head to the sides of the screen but the visual elements will be re-arrange in a different way as if the viewer is looking at the screen from another place or angle. Two key words should be mentioned when it comes to viewpoint control. An *Overview* of a data set is very important for analysis tasks. Given an overview, the analyst may find out prominent property of the whole data set. For instance, one can tell the location of the most expensive buildings in the city by looking at a special drawn map about the prices of buildings, encoded with color. This can be considered as extra information which one cannot acquire so quickly by checking the raw data. In some cases, the original data set is too large to be shown in the limited area of a screen. By panning, probably with a scrollbar, the view gets to see all parts of the set because the viewpoint has been changed. Only with overview, decisions that can be made based on the general shape of the data set are limited or even inappropriate. Hence, detailed information is necessary to avoid this problem. Windows other than the one displaying the overview can be used to show the details of a certain part of the data according to the user's wish. In this case, it usually takes him more efforts to comprehend the chosen content. Details are what the viewer thinks about, generates a mental image based on and finally gets insight from.

Therefore, details cannot bring out a better understanding of some data as quickly as the overview does. In addition, switching between overview and detail may take place when the view forgets about the general situation of the data set while he is checking some data item's details. That's to say, this technique cannot best keep the mental image in the viewer's head. The overview will be lost when he looks into some details in another window. Unfortunately, overview & detail will be used in our project to realize the interchange between the hierarchies of those source file and their corresponding contexts. However, improvements shall be made to alleviate the harm brought in.

The third class of ways of making visualization fanciful is by distorting the graphical elements on the screen. *Focus & Context* is the topic that information visualization people work on mostly. It requires that the focus part or foci be highlighted while its context is maintained. More specifically, when the user selects one data item on the screen, it should stand out other elements in some way to show its importance. At the mean time, the properties of those data items or the environment in which the foci exist should remain the same before the highlighting as much as possible. An example is provided by the so-called perspective wall, illustrated in Figure 2.6.

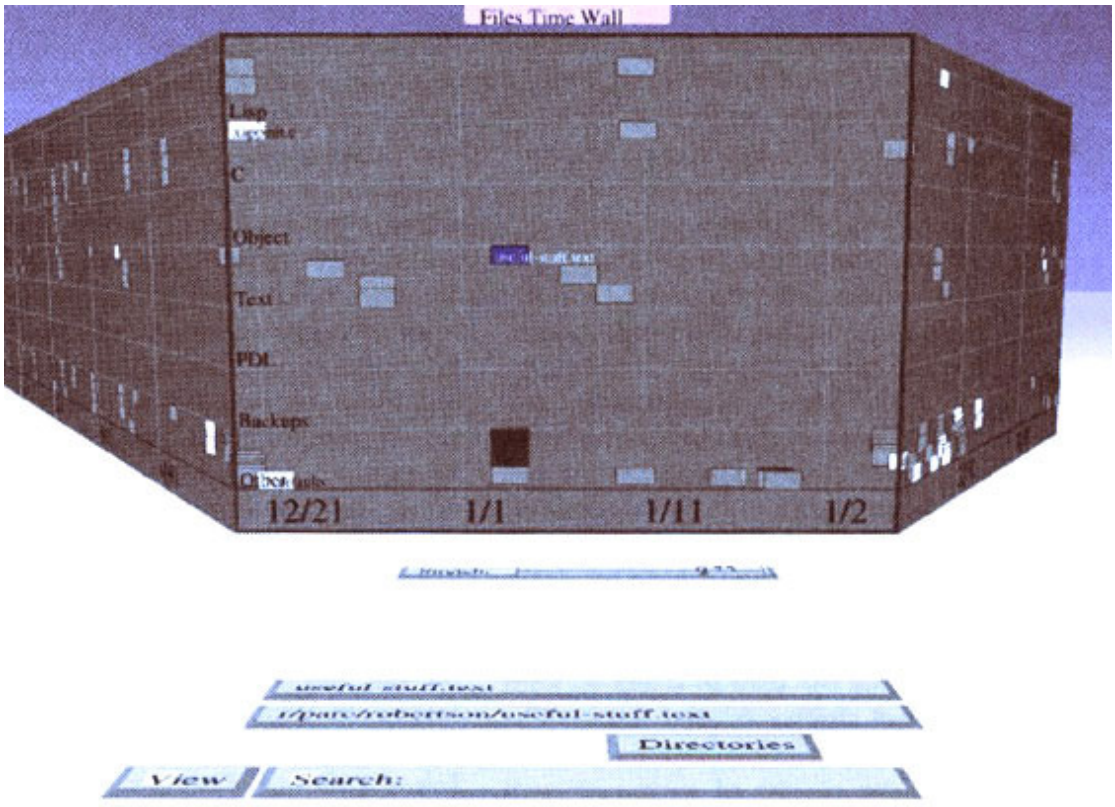

Figure 2.6 Perspective Wall, taken from [5].

As you can see, the original data set is all the files along with their time properties on a hard disk. Due to the fact that this wall is too wide to be displayed even on the whole screen, the two parts which are on the left and right side of the wall have been

folded in such a way that a perspective view is formed. As for which part to be folded, it is up to the viewer's interest. The middle section of the wall is the viewer's focus. Files along with their visualized attributes can be clearly traced in this part. However, it is not the same case for those on the sides. Though one can tell that there are also some files there, it is not possible to look further into it if the viewer does not change the focus part. By distortion, not only large data sets can be visualized, but levels of interest in different data items are distinguished so that viewers can know more about those of importance and temporarily ignore the trivial cases of the moment. Another advantage of this technique compared to overview & detail is that the context of the focus point has been remained. In this case, the viewer does not have to switch between different windows to remind him of the overall property of the data set while he is checking some detailed information as aforementioned.

So far, the main steps of information visualization reference model have all been introduced. There is still an important follow-up phase of this model, that is, the user's interaction with the application. The purpose of using a visualization application is to find insight that the original data set cannot provide directly. Apart from the re-presentation and re-organization of the data, extra and usually helpful information will be acquired through various tasks that a user can perform providing this visualization tool. If the interaction part of the tool is well and vigorously implemented, the user can surely explore the data better and more thoroughly, which usually leads to new discovery of the underlying information. On the other hand, if interaction the user can make with the visual element is limited some valuable messages hidden underneath cannot be exploited, thus the maximal usage of information visualization can by no means be achieved.

2.3 Representation

As mentioned in the previous section, the critical procedure during information visualization is visual mapping. To achieve the transformation from raw data to visual elements, one has to find appropriate approach to represent, or in a simpler way depict those texts and other kinds of data. Before worried about effective way of mapping, it is wise to know further about the things that are supposed to be visualized. In this section, we will talk about the types and complexity of raw data as well as an excellent approach of visualizing relations among data items, which we will make use of in our implementation.

2.3.1 Data Types

Generally, there are two kinds of things that appear in the data set. *Values* are the most common element in a data set. It precisely shows the status of a data item or describes one of its properties. In Nightingale's example from the beginning of this chapter, she used sectors to illustrate the death rates in different hospitals. The bigger area one sector has, the larger number of patients died in that hospital.

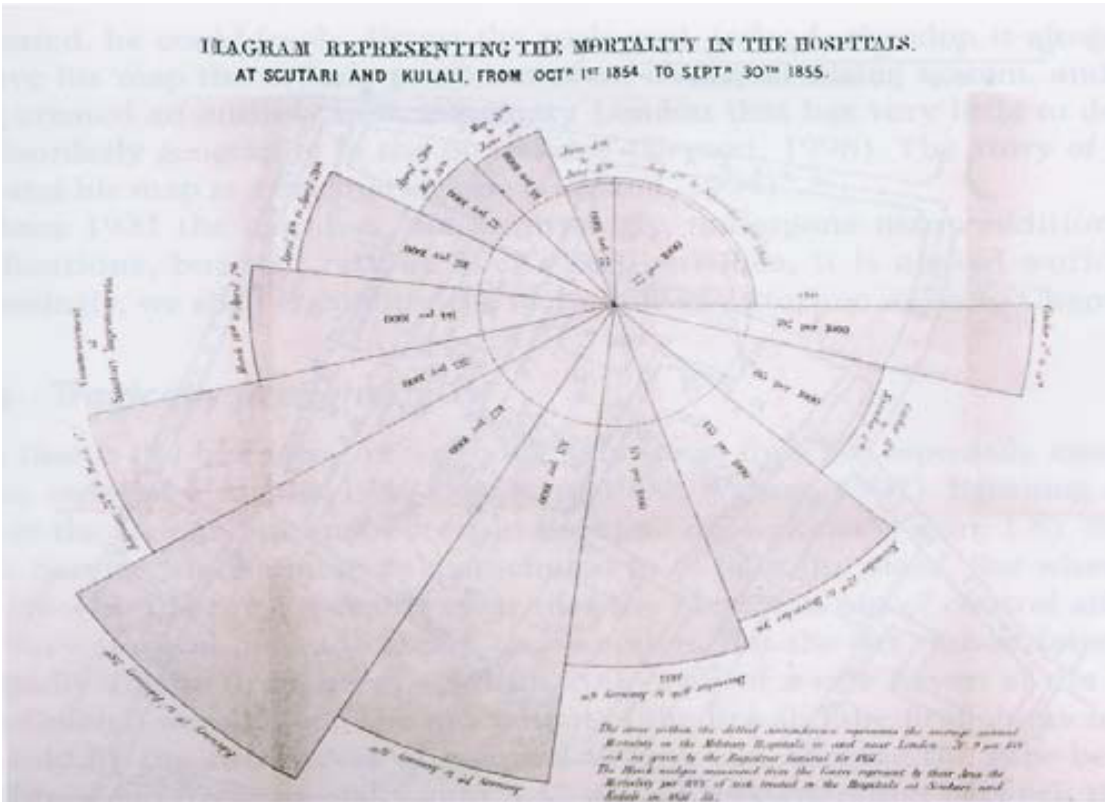

Figure 2.7 Nightingale's Diagram, **Source**: *Nightingale (1858),* taken from [2].

Another example about visualizing values is from the estate field. Prospective clients in some case want to know exactly how much an apartment will cost them. Therefore, visualization people have to be ready to show this kind of precise values on the screen. Apart from the individual price of an apartment, it is possible that some clients like to know the range of the price for the accommodations in a certain area. In this case, the average price, which is derived from the original data, must be included in the visualization as well. It should allow the user to get to know the overall situation of the place in which he is interested by interaction. A sample visualization application is illustrated by Figure 2.8.

The other aspect that we should always pay attention to is the *relation* between

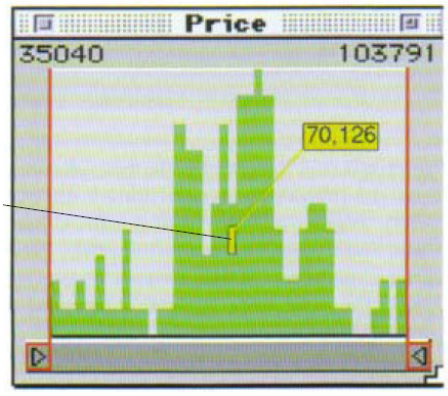

Figure 2.8 Value Visualization, taken from [2]

different data objects. This relation denotes certain association and connects as well as organizes all the data items in such a way that they can form an integral set. Also, relations between two or more objects often reveal something extra that cannot be detected when one looks at the data items alone. Extreme cases are those data only make sense when their relations are clear to the analysis people. There are many ways to represent relations. The commonest approach would be tree representation in which data items are denoted by nodes and relations between them are conveyed by edges. Tree representation has found its application in many different areas. In the world biology, all creatures can be classified into some kind of species. Within one family of animals, the taxonomy can be illustrated by a tree that depicts the inherent relations between different members of the family. Also, tree representation can be an excellent approach of visualizing human heritage. By looking at a family tree, one can easily tell who is whose son with the help of descriptive labels and some additional information such as how many generations there have been in this particular family and whether it has an increasing number of member or it is the other way around. When it comes to business, many companies have brought in the organization map to make its inherent hierarchy more intuitive. In the application illustrated by Figure 2.9 below, interaction is also implemented in order that the view can focus on one member of the faculty and see more details of his/hers. As it can be seen, the focal staff of this large-scaled tree is named Stuart Card whose “cube”, which is a graphical element representing him, is highlighted with the color of blue. While his cube is enlarged so that there is room for his department and position to be shown, some other cubes have been scaled down even to the extent of invisibility.

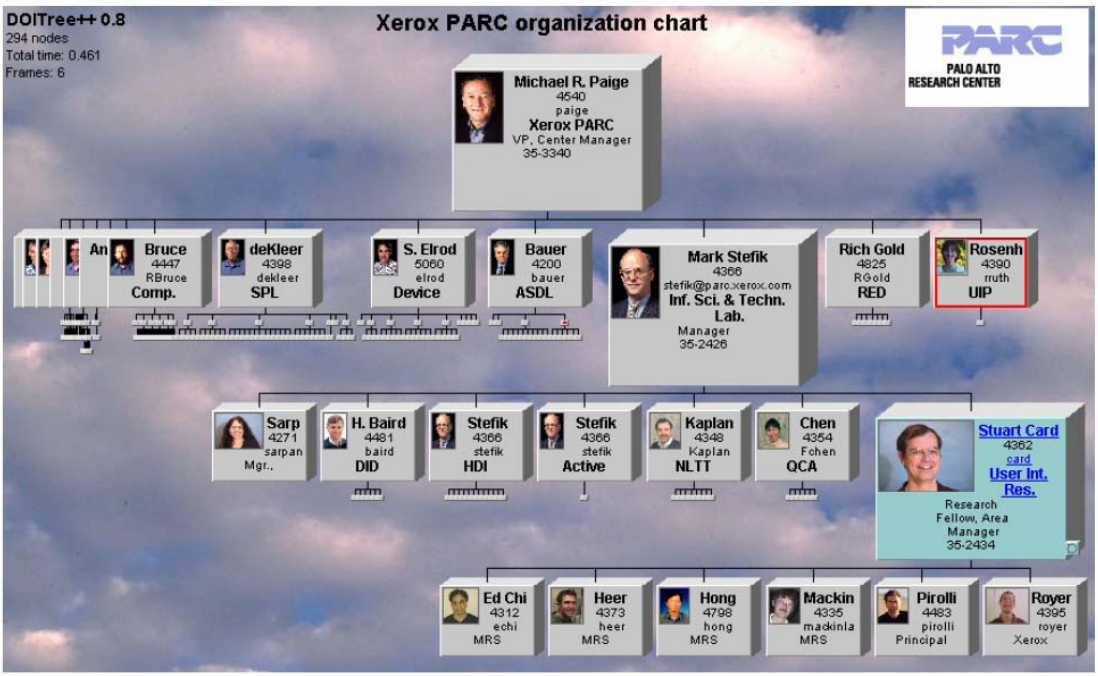

Figure 2.9 Tree Representation of A Company's Hierarchy, taken from [6]

2.3.2 Treemap Representation

Since relations between data items can be of the essence, a lot of approaches have been developed to visualize structural or hierarchical associations. *Cone Trees* map a complex 2D tree into a 3D one which shapes like a cone. The children of a node are evenly distributed on the bottom circle of a cone and the viewer can rotate the tree in other to see those descendants that are originally at the back of the screen. *Degree of Interest* trees, one of which has been illustrated in the previous sub-section, clearly demonstrate the structure of the company's organization and make view-adaption according to the user's interest. Among those creative and intuitive approaches, one approach depicts hierarchies in such an excellent way that it has become the most popular method of visualizing hierarchical information nowadays, that is, Treemaps [7]. As you will see soon, treemap representation is rather good at demonstrating relations between data items. Therefore, we decide to take advantage of it in our project in order to visualize the structures of read-in Java source files.

Treemaps are initially introduced by Johnson and Shneiderman in 1991. In general, it is an algorithm that partitions the display space recursively with rectangles in such a way that the resultant visualization reveals the hierarchy of all the data items. Those rectangles are a brand new form of the nodes in a traditional tree representation. Apart from the hierarchy, the properties of the nodes' are also an indispensible aspect of visualization. Since the data items have been converted into rectangles on the treemap and each rectangle surely has a certain area, the attributes of a node can be readily encoded within that area [8]. As Johnson suggested, treemap is a space-filling approach which can make effective usage of the limited space on the screen. Traditional tree representation sketches the nodes in a layered manner. Typically, the whole tree will shape like a triangle. In this case, a large portion of the display area is left blank. However, when treemap is used the screen area will be filled by as many rectangles as it can hold, which means no space is wasted thus this algorithm does achieve a very effective utilization of the display area.

Generally, there are two kinds of treemaps that are common these days. Non-nested treemaps are the earliest layout introduced by Johnson and Shnerderman. In this layout, all the leaf nodes are explicitly drawn out on the map while intermediate ones cannot be recognized intuitively. Due to this short-coming, interactions are limited to leaf nodes. The viewer can only manipulate leaf nodes to access the properties of theirs. Therefore, this non-nested algorithm is only effective when the information on leaf node is highly-prioritized. However, if interaction allows the user to create another view based on the selected leaf node, the presence of its ancestors becomes possible, which can lead the viewer to the attributes of them.

Another improved layout is called nested layout in contrary to the former one. In this layout, the rectangles of the descendents no long take up all the area of their parent, but leave a little space so that the ancestor has room to show some of its properties within its own territory. In this case, people can easily tell that relationship between a node and its parent. In addition, interaction is possible on intermediate nodes as the user can actually for example click on them. Although a major improvement has been made and still the whole screen area is utilized, the rectangle

for each leaf node becomes smaller that before because of its parent, which means the room for leaf nodes to show their properties is reduced [8]. Assume that information on each child is rather intensive, it is impossible for the diminished rectangle to hold all the values at one time. In this case, another view of the focus node is still necessary although access to intermediate nodes is feasible on the original overview.

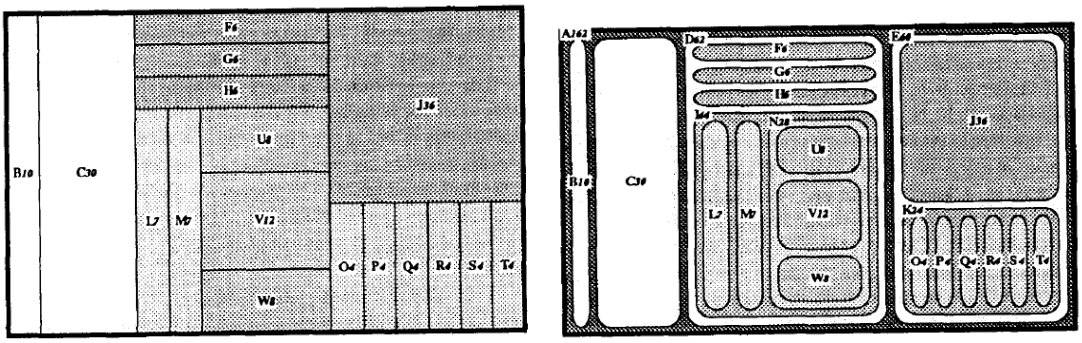

Figure 2.10 Tree-Map and Nested Tree-Map, taken from [8].

As illustrated above, treemaps can depict the hierarchy effectively and excellently no matter nested or not. It gives an intuitive overview of the whole data set, which tree representation can hardly achieve, especially when the hierarchy becomes complicated. Attracted by the prominent feature, lots of applications favor this layout in order to make their data sets better understood. *New Groups* is a well-known success of treemap applications. In Figure 2.11, all the articles, posts and essays are grouped by their concerned fields. The number of the writings in each category is enc-

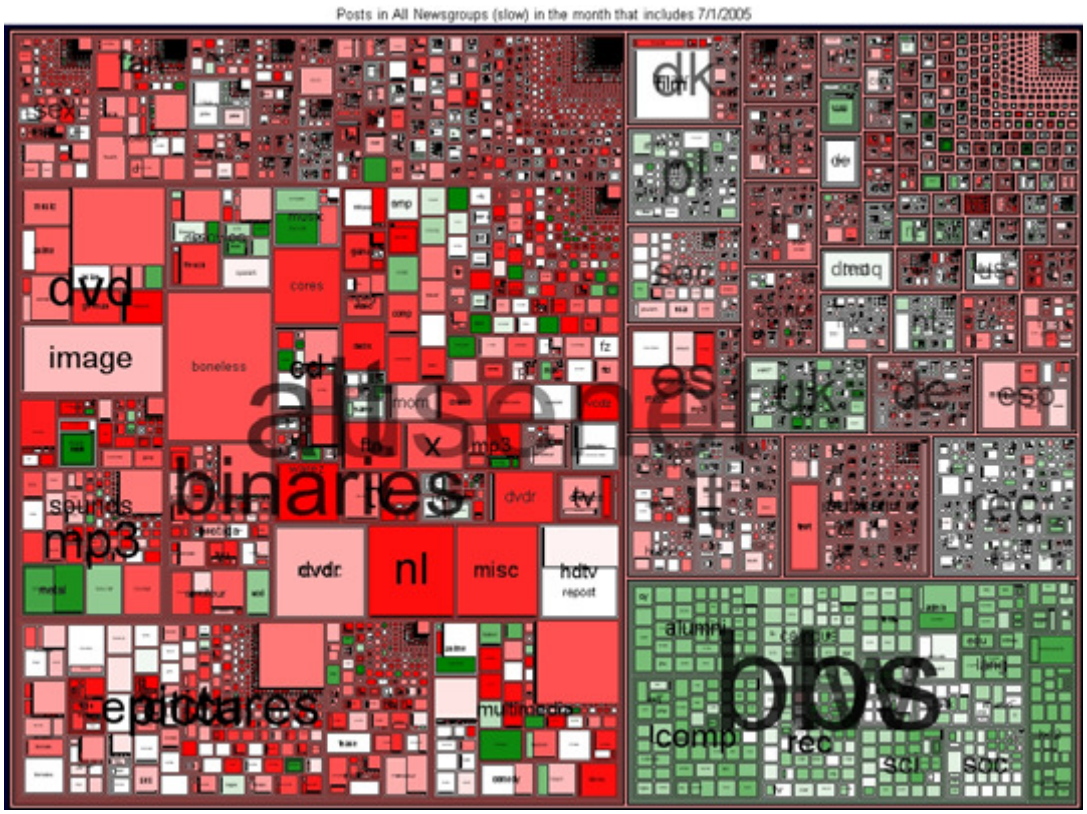

Figure 2.11 News Groups, taken from [6]

oded by the size of the rectangle. The more posts there are, the bigger the corresponding rectangle is. Also, colors from red to white and then from white to green can be interpreted as the popularity. If the readers become more interested in a particular area over months, the greener the rectangle that represents that area will be.

As it can be seen from above, treemap can provide a very good overview of hierarchical data. Since each Java source file has a certain structure which can be represented by an abstract syntax tree, visualizing the structure with a treemap is undoubtedly a wise and effective approach. In our thesis project, providing the viewer with the hierarchies of all input files is one of the first-prioritized requirements. Having an overview of the files' structures not only helps the user get an overall impression on the data set, but also makes it possible identify duplicated programming elements without even looking at the source code. This kind of idea will definitely save the user's efforts on doing his analysis task and the research time is reduced. Since intermediate nodes of an abstract syntax tree, i.e. a rectangle representing a method, may be accessed by the user, interaction on these entities should be provided on the treemap. As aforementioned, with nested treemap, hierarchies are more obvious and easier to be recognized. The visualization may be more effective if this kind of treemap is the base of our implementation. However, non-nested treemaps can also make accesses to intermediate nodes possible by bringing in another view of a treemap whose node has been selected. Both approaches are applicable in case that we implement our own treemap algorithm. Due to some scheduling problem, we decide to use an off-the-shelf tool kit to help us with the visualization for the purpose of saving time. Unfortunately, this tool kit, which is very powerful when it comes to visualization, only provides an algorithm that generates non-nested treemaps and this is apparently not the optimal solution to our task. Therefore, a brand new approach which can be as effective as nested treemaps although non-nested ones are used has been devised to improve our visualization under such circumstance. The concrete design will be explained in chapter 4.

3 Related work

This chapter presents some related works. Some similar tools that are relevant to our clone detection tool have already been published; they have their own outstanding points. In this chapter, we will simply list some of those tools and briefly compare them with our tool that is going to be presented in following chapter.

3.1 Clone Detection Results Plug-in

This project, which is presented at [17], looks into an alternative visualization method by extending the AspectJ Development Tool (AJDT) Visualiser plug-in that was originally used to display aspects in program modules. The freely available Java version of CloneDR^{TM} was integrated into Eclipse through a customized plug-in. In addition to implementing an extension of the Visualiser, this plug-in also utilizes Eclipse's features to provide an enhanced interface for CloneDR. The process is depicted in the following figure.

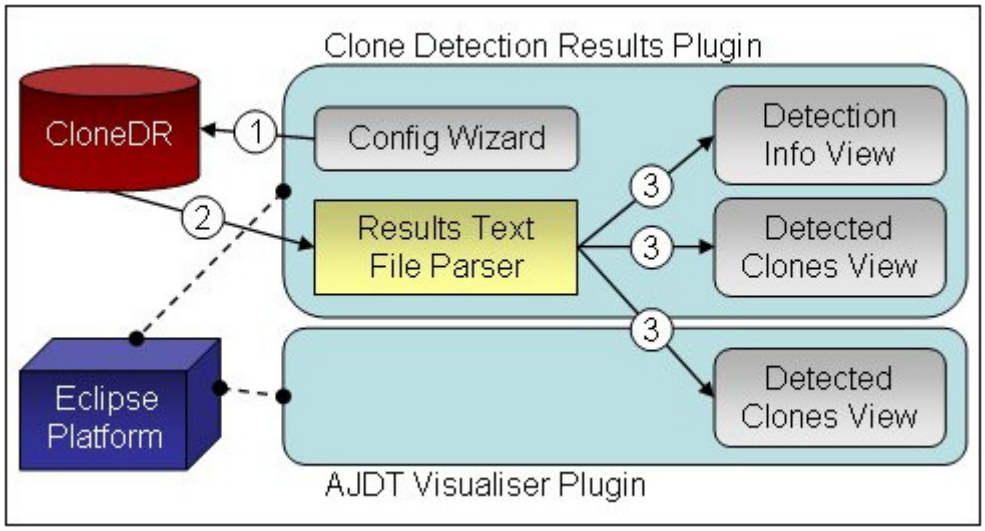

Figure 3.1 Plug-in architecture and process, taken from [17]

Wizard pages in the Eclipse plug-in assist users to determine the configuration settings for the clone detection procedure. These settings are passed to CloneDR (step 1). CloneDR will run and upon completion of its detection process will generate a text file containing its detection results. This file is parsed by the plug-in (step 2) and the information is passed to three different views. One view displays the detection process information and statistics. The other two views display the clones that were detected through two types of representation: a text listing and the Visualiser view.

We have to acknowledge that this plug-in is very powerful in some aspects. But it can do nothing without CloneDR and Eclipse. It has strong dependence on CloneDR and Eclipse. For this point, we believe in the superiority of our clone detection tool over this one, since it is independent of any software. Once all dataset that are required by our tool are given, user can find similar or identical parts easily with the help of our tool.

3.2 DUPLOC

DUPLOC [23] reads source code lines and, by removing comments and superfluous white space, generates a *'normal form'*-representation of a line. These lines are then compared using a simple string matching algorithm. DUPLOC offers a clickable matrix which allows the user to look at the source code that produced the match as shown in Figure 3.2. DUPLOC has the possibility to remove noise from the matrix by '*deleting*' lines that do not seem interesting, e.g. the **public:** or **private:** specifiers in C++ class declarations. Moreover, DUPLOC offers a batch mode which allows to search for duplicated code in the whole of a system offline. A report file, called a *map*, is then generated which lists all the occurrences of duplicated code in the system.

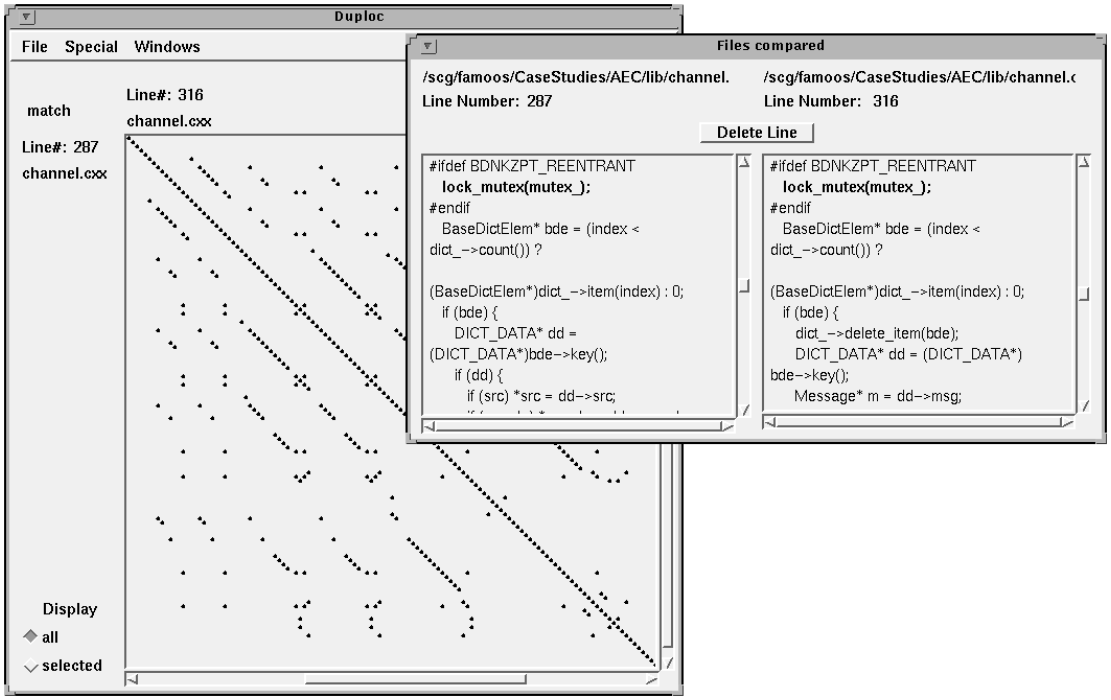

Figure 3.2 The DUPLOC main window and a source code viewer, taken from [23]

Compared to our tool, this tool does not provide a pretty hierarchical structure for each input file; It does not allow to visualize large amounts of data on a single screen; Besides, it cannot tell user what the similarities between many different files are, and it does not provide a visualization that is as splendid as ours, while our tool that is presented in this thesis provided these functionalities.

3.3 SeeSoft

As presented in [19], **SeeSoft**, and related system SeeSys, visualize various textual aspects of evolving large and complex software systems. Such aspects involve software complexity metrics, number and scope of modifications, number and types of bugs and dynamic program slices. Managers of such system development projects need to be able to gain overview information of the system development activity. Analysts need to know how to restructure the code during the next development cycle.

Testers need to know what has changed in order to test the new features and bug fixes.

SeeSoft is implemented using the information from version control systems. These systems keep track of every single line of code, including the dates of changes and reasons for changes and the developer who changed the code. The motivation behind SeeSoft is to display as much information as possible by using pixels to represent information, and to use as many pixels on a screen as possible.

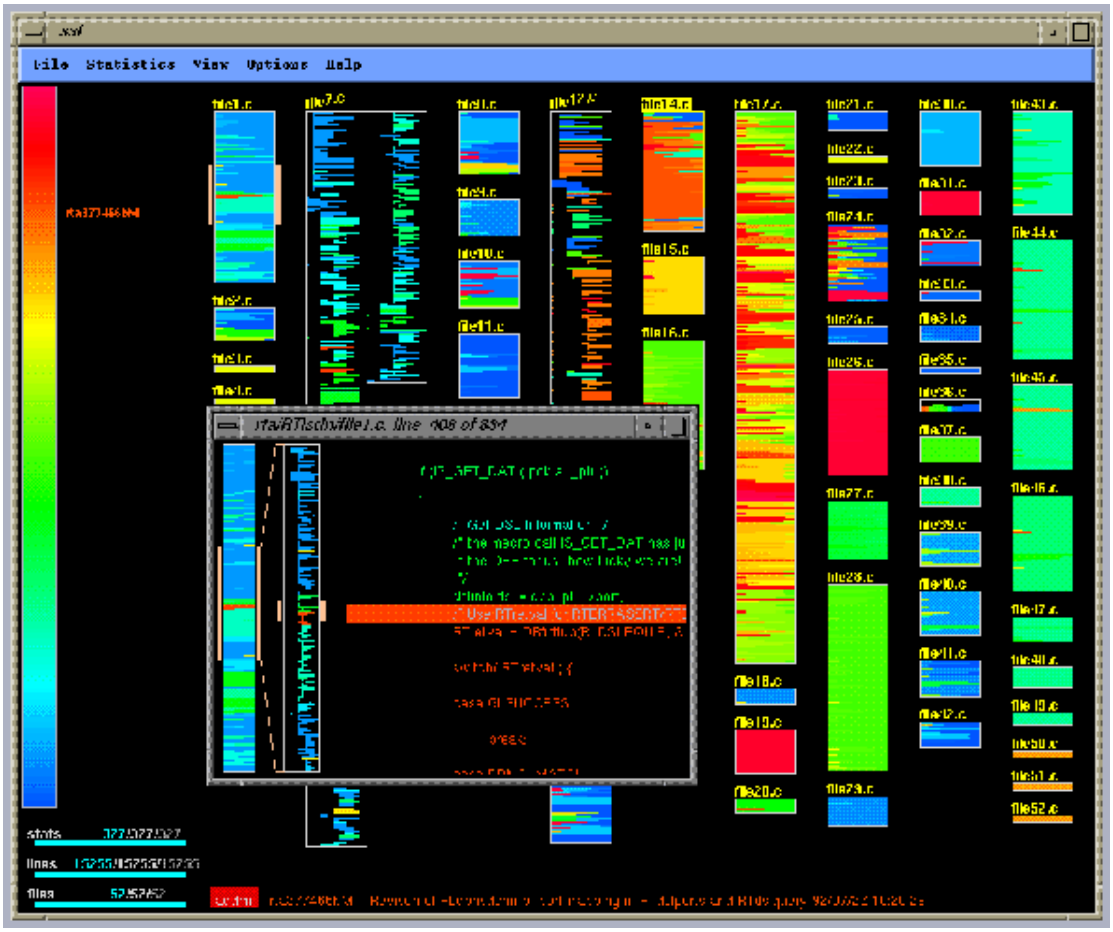

Figure 3.3 Various screen shots of SeeSoft and SeeSys, taken from [19]

But this software does not show equal or similar text parts directly. Or rather, it does not show how many percentage of two compared Java file are similar to each other. Besides, our tool not only can visualize and compare Java files, and finally give users a nice result of comparison, but also may compare and discover plagiarism in different types of documents (future work), our tool that allows us to find similar text parts in different documents could help us in this situation.

3.4 Radial document visualization

The tool Radial document visualization, which is presented in [20], is a data visualization method that is based on “sunburst”[14] for representing document content by a radial space filling layout technique. "docuburst" aims to provide a more intuitive abstraction than those developed through statistical techniques. The root

node is shown as a circle. All other nodes are assigned to a sector of an annulus with angular width which is part of the parent node's width, depending on the amount of word occurrences. Highly colored nodes have many occurrences, while almost transparent nodes have few occurrences.

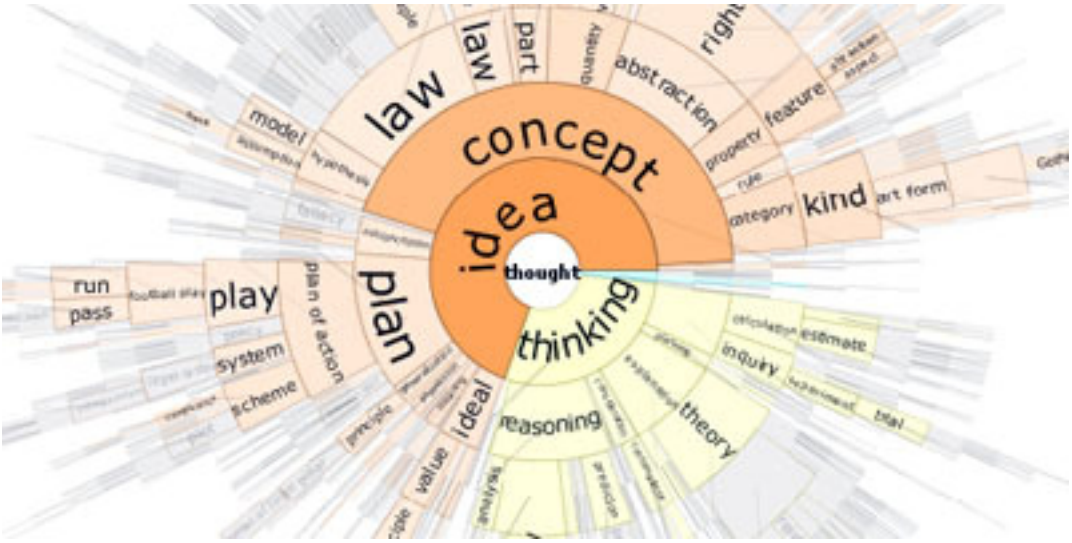

Figure 3.4 Example of Radial document visualization, taken from [20]

This tool provides a great layout of documents, which is very impressive to me. But obviously, compared to our tool, this tool is not so powerful enough to get different documents compared. Actually, it only can visualize one document each time. But this way to visualize document is really a good way for us to learn or simulate.

3.5 IN-SPIRE

IN-SPIRE [21], powerful information visualization software developed by Pacific Northwest National Laboratory, can give people the ability to see something different in the data they already have.

IN-SPIRE can quickly and automatically convey the gist of large sets of unformatted text documents such as technical reports, web data, newswire feeds and message traffic. IN-SPIRE can handle real-time data by adding new documents as they arrive. It also processes foreign language data and provides robust support for translation. By clustering similar documents together, this Windows-based software unveils common themes and reveals hidden relationships within the collection. A screen shot of this tool is as follows:

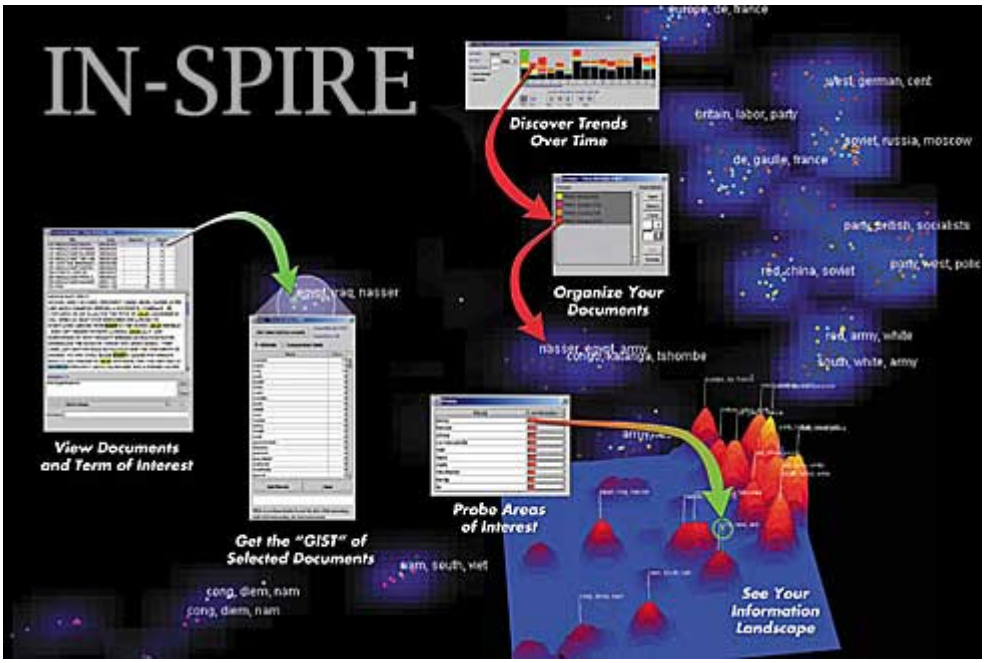

Figure 3.5 Screen shots of The IN-SPIRE discovery tool, taken from [21]

This tool gives people the ability to see something different in the data. We have to acknowledge its powerfulness. But it is not specially designed for comparing Java files (software system). When it comes to the analysis of software clones, our tool will be more outstanding and show more available and latent superiorities**.**

3.6 Other Related Tools

There are several clone detection tools; they are listed in [22]. But none of them has a nice layout of sources (e.g. Java files) that are analyzed, and none of them provides a nice visualization of the result. The visualization of the result provided by our tool will become more fine-grained step-by-step (so-called semantic zooming) depending of the user interaction and the underlying data set. Plus, these tools only focus on the duplicate codes (like C, C++, Java and so on). When it comes to the documents of other types (e.g. XML file), they cannot do anything. Our tool can solve this problem; this functionality will be implemented in future work.

4 Visualization Approach

This chapter presents some visualization approaches that are used in our application, for example, how to visualize a Java file, and in which way we are going to visualize the identical or similar parts between many different Java files. There are several critical requirements that the tool is supposed to meet. First, the tool should show the similarities between single documents. Second, it should show the hierarchical structure of the documents so that an overview of the whole data set is available to the viewer. Third, it should provide different levels of abstraction with regard to the hierarchies. Forth, accesses to the code should be possible. Fifth, powerful interaction techniques should be offered to help the view with analysis work.

4.1 Visual Mappings

Firstly, we are going to present the way how to represent Java files, both the overview and detail of Java files. Since every Java file can be represented by Abstract Syntax Tree, we are going to use Treemap to represent Java file. Here comes the detailed approach of visualization.

4.1.1 Treemap Layout (Overview)

Treemaps are a space-filling algorithm that represents nodes as boxes on the display, with child nodes represented by boxes placed within their parent's box. The treemap Layout algorithm is used to display the tree structure better than the traditional node-and-link diagram, since it can fully utilize the limited space and give a good overview to users. Here is an example visualization of our application. The following two screen shots are the overview of loaded Java files, 56 treemaps are displayed in the frame showing an overview, which means 56 Java files have been analyzed in this case.

Figure 4.1 shows that nodes are shaded according to their size of program elements they represent. The darker the node is, the larger its size is. Figure 4.2 shows that nodes are shaded according to their depth in the tree. The darker the node is, the deeper the depth is (assume the depth of root node is 0). This functionality provides a good overview; size and depth can be switched between each other by clicking the menu “Layout” and choosing the menu item *Size* or *Depth* according to user’s interest. Plus, user can choose another Treemap Layout, in which the **area of a node** in Treemap depends on the attribute *size* of the node.

Squarified Treemap Layout algorithm is the mainly used for displaying every TreeML file (or Abstract Syntax Tree). This particular algorithm is taken from [12]. For more information on treemaps in general, see [13].

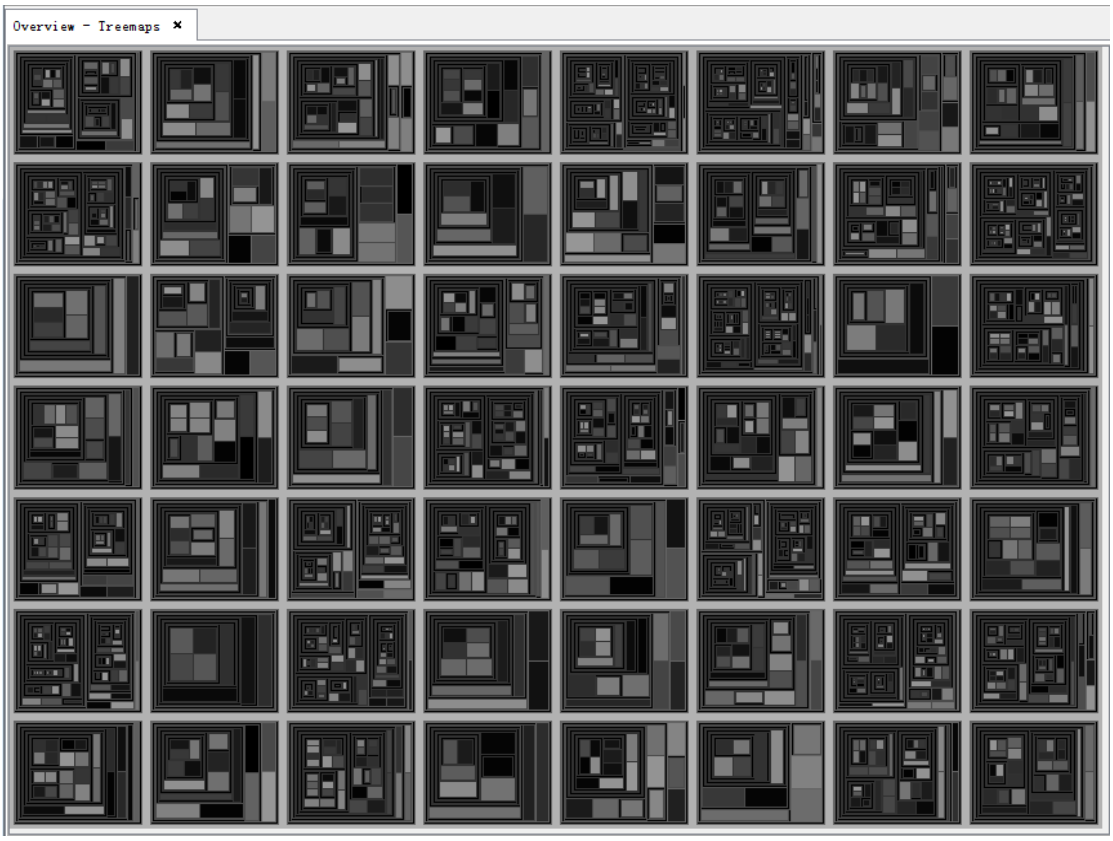

Figure 4.1 Example visualization (Shaded according to Size)

Figure 4.2 Example visualization (Shaded according to Depth)

We developed another Treemap based on the Squarified Treemap Layout algorithm. In our Treemap algorithm, the **area of a node** in Treemap depends on the attribute *size* of the node. Color and Label of the nodes are added based on Squarified Treemap. The related codes are given in Appendix A.2.

For the Class *LabelLayout, FillColorAction, BorderColorAction* and other more codes in detail, please refer to Appendix A.2. This Class ***LabelLayout*** simply sets the positions of labels. Labels are assumed to be DecoratorItem instances, decorating their respective nodes. The layout simply gets the bounds of the decorated node and assigns the label coordinates to the center of those bounds. The codes, which are used to create the labels as decorators of the nodes, are as follows:

```
m_vis.addDecorators(labels, treeNodes, labelP, LABEL_SCHEMA);
```

For filling the color of nodes, the most related important Class is ***FillColorAction***. This Class is used to set fill colors for Treemap nodes. As mentioned above, the nodes are shaded according to their *Size* (of tokens) or *Depth* in the tree.

Class ***BorderColorAction*** sets the stroke color for drawing treemap node outlines. A graded grayscale ramp is used in default, with higher nodes in the tree drawn in lighter shades of gray.

4.1.2 Star Burst Layout (Detail)

Apart from Treemap, the data set will be mapped into another visual structure in our application. This visual structure is based on “sunburst” [14] techniques.

In our application, for instance, if user would like to see the detail of a treemap, he or she could **double click** any node that he or she is interested in. Then, another window will pop up, in which the tree will be displayed with radial representation. The Figure 4.3 shows what it looks like. The main Java codes are as follows, it creates the tree layout action and adds layout schema to nodes. Plus, it creates the filtering for the visualization:

```
StarburstLayout treeLayout = new StarburstLayout(tree);
ActionList filter = new ActionList();
filter.add(fisheyeTreeFilter);
filter.add(new TreeRootAction(tree));
filter.add(treeLayout);
filter.add(new StarburstLayout.LabelLayout(labels));
filter.add(subLayout);
filter.add(textColor);
filter.add(nodeColor);
filter.add(nodeStrokeColor);
m_vis.putAction("filter", filter);
```

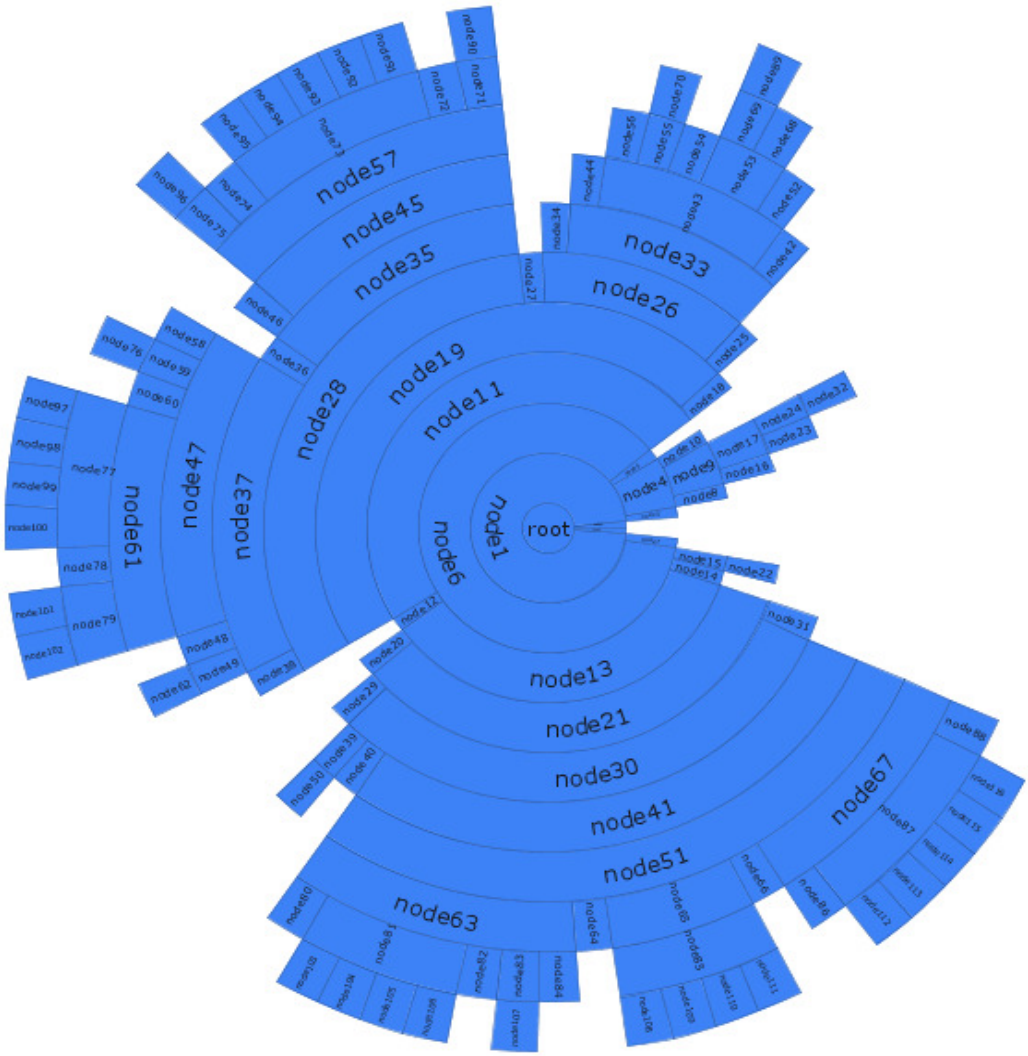

Figure 4.3 The radial visualization of a tree.

The Class *StarburstLayout* is a *prefuse.action.layout.graph.TreeLayout* instance that computes a radial space filling layout, laying out subsequent depth levels of a tree on circles of progressively increasing radius. It is based on radial layout implementation for node-link diagrams by Jeffrey Heer [15].

The code for displaying and interacting with radial, space filling trees in PREFUSE is open source, and is available for downloading. The code is distributed as a zip file and can be imported into Eclipse. It is dependent on the PREFUSE information visualization toolkit. In our application, we changed the code a little bit to make it more fit to our tool. The most important codes that we added are as follows, it makes this radial visualization firstly focus on the node you clicked in the overview frame. Class `KeywordSearchTupleSet` provided by PREFUSE is used to implement this functionality. For more information on this radial document visualization, please refer to [16].

```
//show the node you clicked first if it is not null
if(ItemGlobalKey!=null)
{
```

```
    KeywordSearchTupleSet mysearchTS=(KeywordSearchTupleSet)
vis.getGroup(mykeywordsearch);
    SearchQueryBinding sq1 = new SearchQueryBinding((Table) vis
                .getGroup(treeNodes), "GlobalKey", mysearchTS);
    mysearchTS.search(ItemGlobalKey);
}
```

4.2 Interaction Techniques

In our thesis, we are supposed to make our visualization tool be capable of detecting the similarities of top level among those analyzed ASTs (Abstract Syntax Trees) or the identical parts among these nodes.

It makes no sense to show all similarities between the nodes distributed in many different documents, because there are too many, and that information is not important for the user. But the important things are IDENTICAL parts between the nodes of many different ASTs. Typically, you will have about 100-150 identical parts, depending on a threshold value. That value says that only identically parts will count if a specific number of lines are detected. Thus, a value 1 means that each identical line is counted; a value 5 means that only 5 identical lines will be counted. As a result, we may have **two sliders**: one for the **SIMILARITY** of the top level; and one for that threshold value of **IDENTICAL Lines**.

4.2.1 Interaction with Top Level SIMILARITY

Here is the example showing the way our visualization tool deals with the **SIMILARITY** on top level.

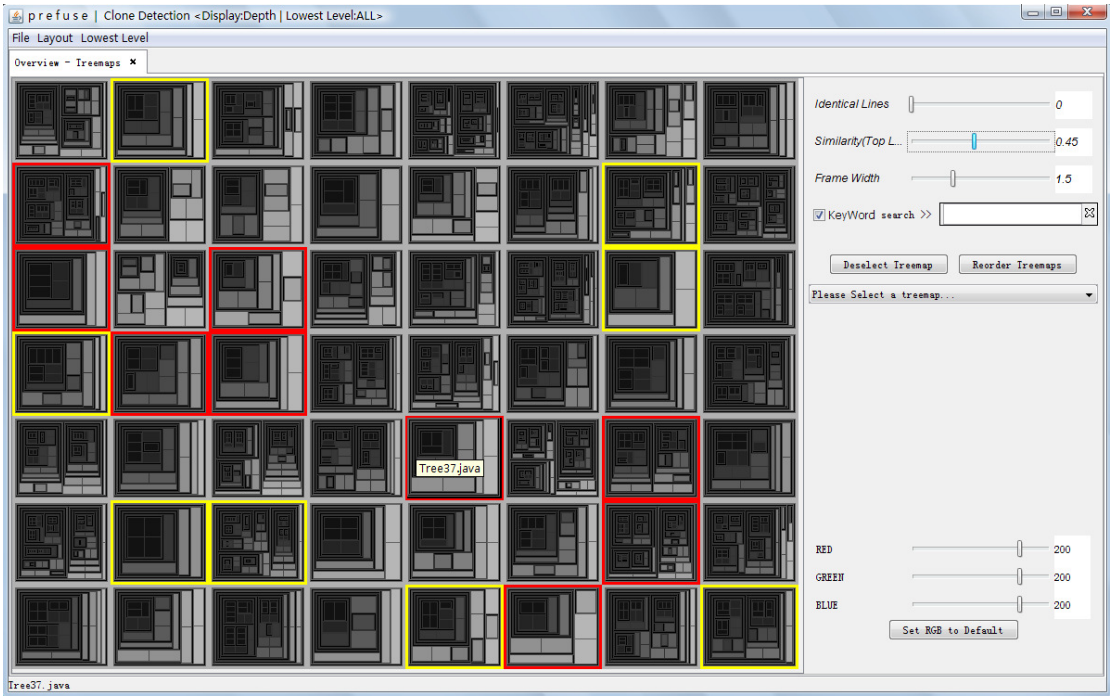

Figure 4.4 Example of showing SIMILARITY of the top level

Figure 4.4 shows the visualization example when we set value of the slider of SIMILARITY to 45%. The borders of all the trees which may be 45% similar to other trees are highlighted in red. It is easy to see that the border color of some Treemaps is yellow in this figure. That is because when the cursor is over a highlighted Treemap *T,* the borders of those treemaps which are more than 45% similar to *T* will become yellow. Here are some source codes for SIMILARITY Slider, Class `JValueSlider` is provided by PREFUSE:

```
// similarity Slider
JValueSlider similaritySlider = new JValueSlider(
       "Similarity(Top Level)", 0.0, 1.0, 0.0);
similaritySlider.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e) {
           // do something here
           }
       });
```

For more information about how it is implemented and more source codes in detail, please refer to Appendix A.3.

Here is another functionality that is suggested by our potential "client" during the development. The button "Order" which is at right side of the screen is used to "re-order highlighted treemaps according to the number of files that are similar with each file given the current similarity value". When moving the cursor over this button, the tool tip will show up to tell you what this button is for. This button can be of great use. For example, the treemaps with red border are spread arbitrarily in Figure 4.4. The user may want to know which Treemap has the most treemaps that are 45% similar to it. But, we cannot tell at a glance. So, the button "Order" will help in this case. The treemap which has the most treemaps that are similar to it in current similarity will be placed at the upper left corner of the screen, and by analogy. Figure 4.5 shows reordered treemaps.

Furthermore, if right clicking on a highlighted Treemap *T*, a menu will pop up, all treemaps that are highlighted in yellow, which also means that all treemaps that are 45% similar to *T*, will be listed on the popup-menu as menu items. The labels of those menu items are the name of TreeML files. A screen shot (Figure 4.6) shows visualization of this example.

Besides, the user may move the cursor from one menu item to another on the Popup-Menu and the corresponding Treemap will be perceived with another color around the borders. And if clicking any of the menu items, another window will pop up to show detail of clicked menu item (the corresponding AST tree) in radiant document visualization. Figure 4.7 is showing this example visualization of a detailed tree.

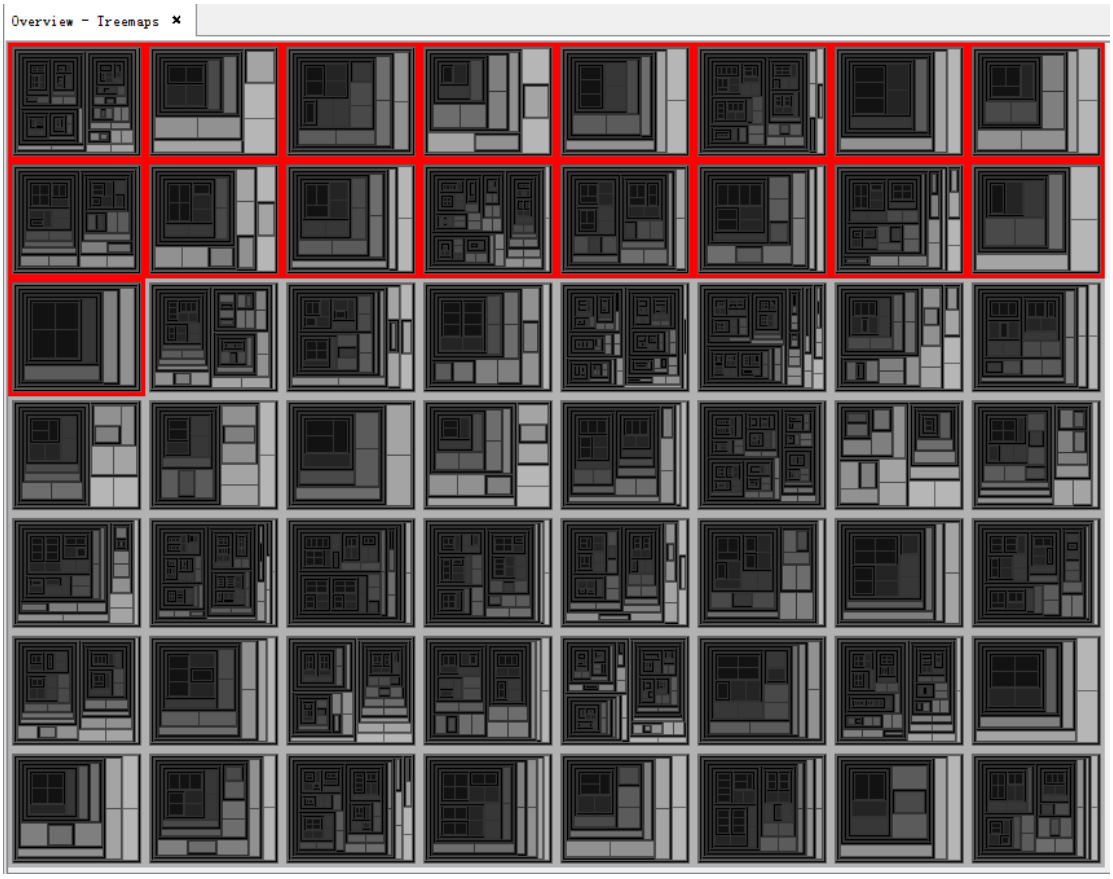

Figure 4.5 Example of showing reordered treemaps

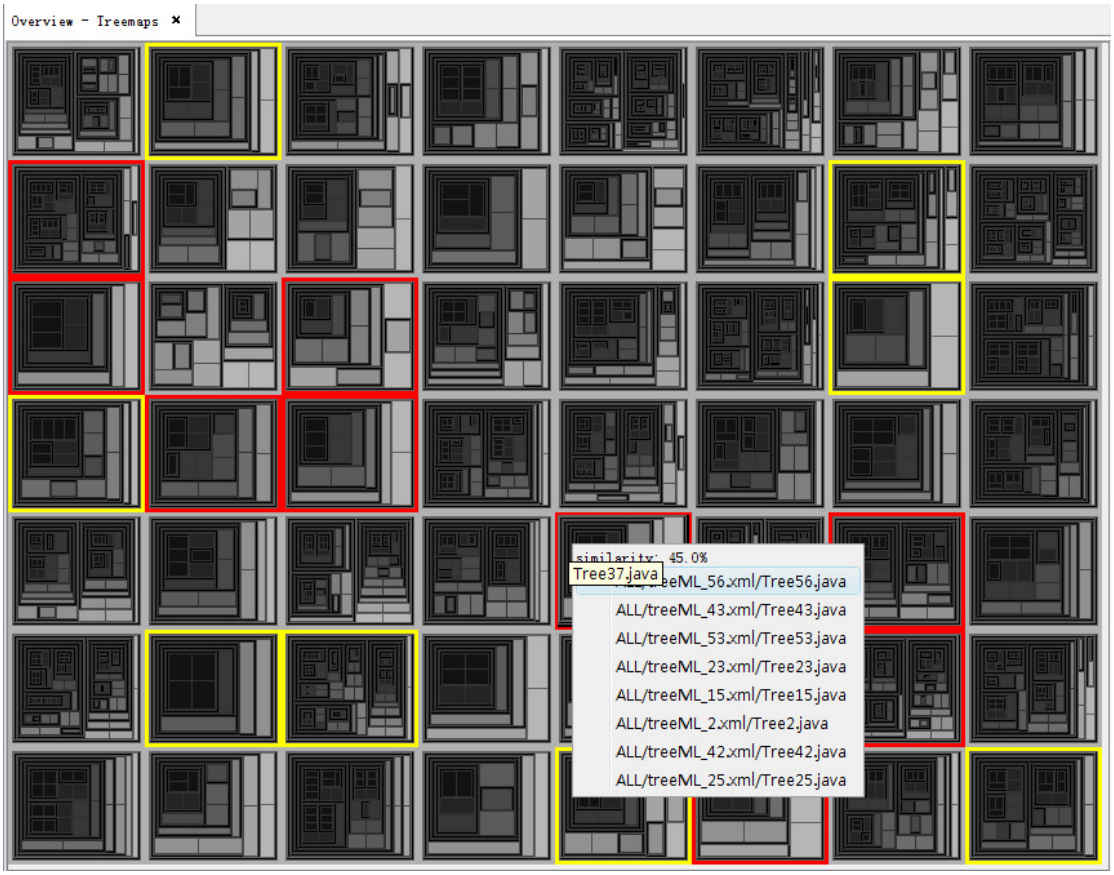

Figure 4.6 Example of showing Popup-Menu (right click)

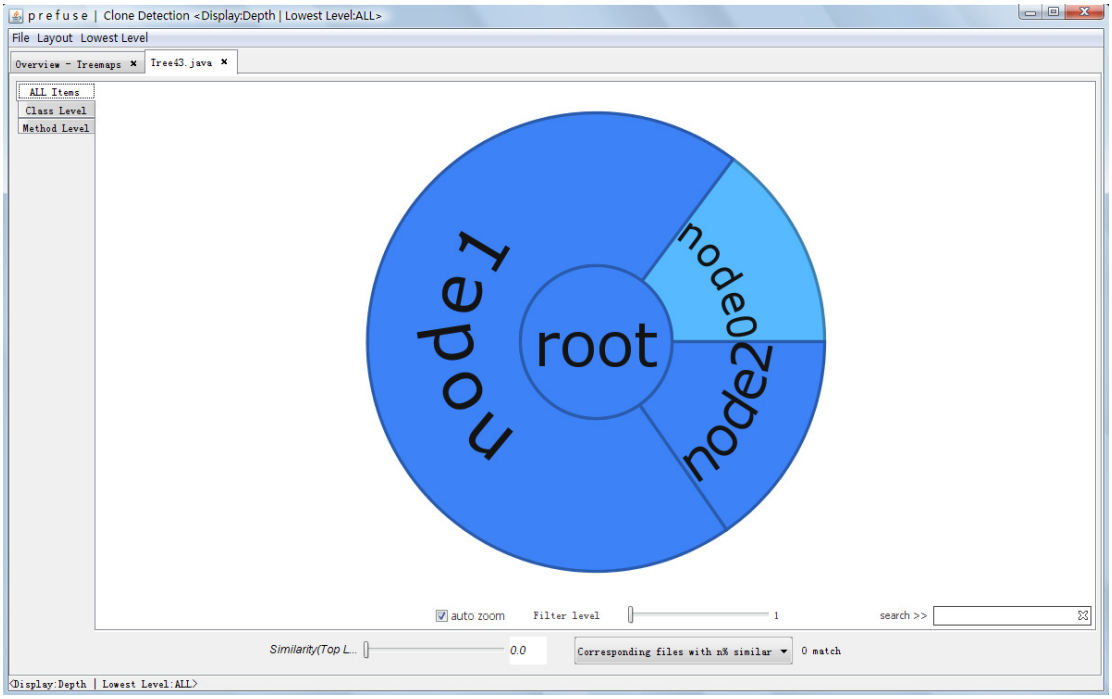

Figure 4.7 Example of showing radial document visualization

In Figure 4.7, we can easily see that there are three levels of this radial document visualization at the left side of the screen. They are *ALL Items, Class Level,* and *Method Level.* The concepts of these three levels are introduced in section 5.3.2 already. User can choose any level that he or she wants among these three options.

Plus, we can see there are a **SIMILARITY slider** and a Combo Box on the bottom of this screen, if we set the **SIMILARITY slider** to a value n% randomly, the files which are n% similar to the current file will be listed in the combo box aside. And if the similarity changes, the files listed in the combo box will change correspondingly.

Selecting a Treemap

To make the user interface more friendly, we add some additional interaction techniques for user to find similar files more easily.

The user may select a treemap that he or she has interests in by left clicking on a treemap or choosing the items listed in combo box on the right side of the window. The border of selected treemap turns into **Green**. Meanwhile, **SIMILARITY slider** is only available for this selected treemap. That means if changing the value of **SIMILARITY slider** into n%**,** the application will make the treemaps, which are n% similar to the selected Green treemap, highlighted in Yellow. The visualization will change as the value of similarity changes. Here is a screen shot showing this technique. In this figure, the value of **SIMILARITY slider** is 45%, so the selected green treemap is more than 45% similar to these four treemaps with yellow border.

Figure 4.8 Example of showing interaction with selected Treemap

Moreover, we can use the combo box or button “Deselect Treemap” to the right to deselect the treemaps. Once there is no treemap selected, the **SIMILARITY slider** is available again for all treemaps.

4.2.2 Interaction with Identical Parts

Firstly, the following figure is the example showing that how our visualization tool visualize the **results** of identical parts between the nodes of many different ASTs.

Figure 4.9 shows the visualization example when we set value of the slider of identical lines to 1. All the nodes that may have more than 1 line in common with other nodes are highlighted in purple. It is not difficult to see that some nodes are highlighted in yellow at that time. That is because when the cursor is over a highlighted node *N,* those nodes which have more than 1 line in common with *N* will become yellow as well. If we move the cursor away, the yellow nodes will become purple again. Here are some source codes for Identical Lines Slider, Class *JValueSlider* is provided by PREFUSE. For more information about how it is implemented and more source codes in detail, please refer to Appendix A.3.

```
// identical detection slider
JValueSlider identicalSlider = new JValueSlider("Identical Lines", 0,
                10, 0);
identicalSlider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
```

```
            // do something here
            }
});
```

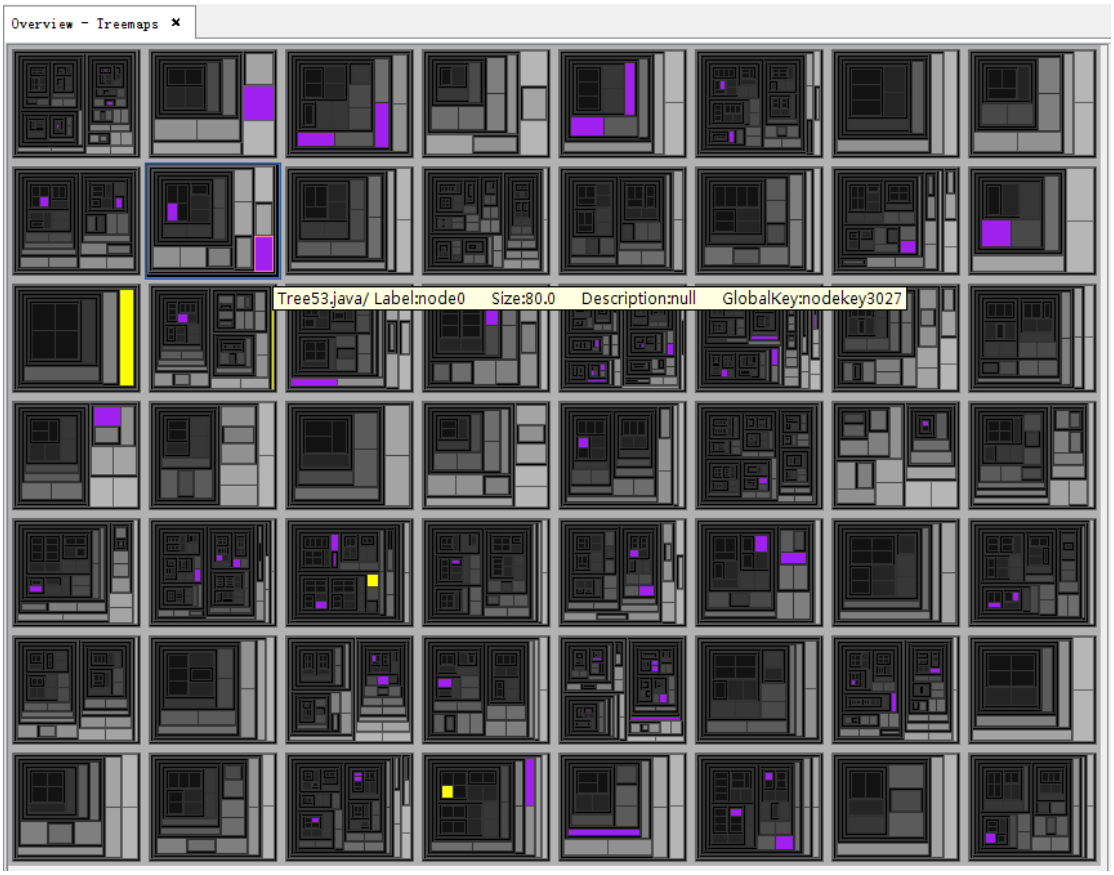

Figure 4.9 Example of showing identical parts between nodes

Furthermore, if right clicking on a highlighted Node *N*, a menu will pop up and all the nodes that are highlighted in yellow, which also means that all those nodes that have more than 1 line in common with *N*, will be listed on the popup-menu as menu items. The labels of those menu items are the labels of the corresponding nodes and the name of treemaps which they belong to. A screen shot (Figure 4.10) shows visualization of this example.

What's more, the user may move the cursor from one menu item to another on the Popup-Menu; the corresponding node will be highlighted explicitly. And if clicking any of the menu items, another window will pop up to show the details of clicked menu item in radiant document representation. And the clicked node will be highlighted in red in both **the radial document visualization** and **the overview frame**. Figure 4.11 shows a visualization of this example.

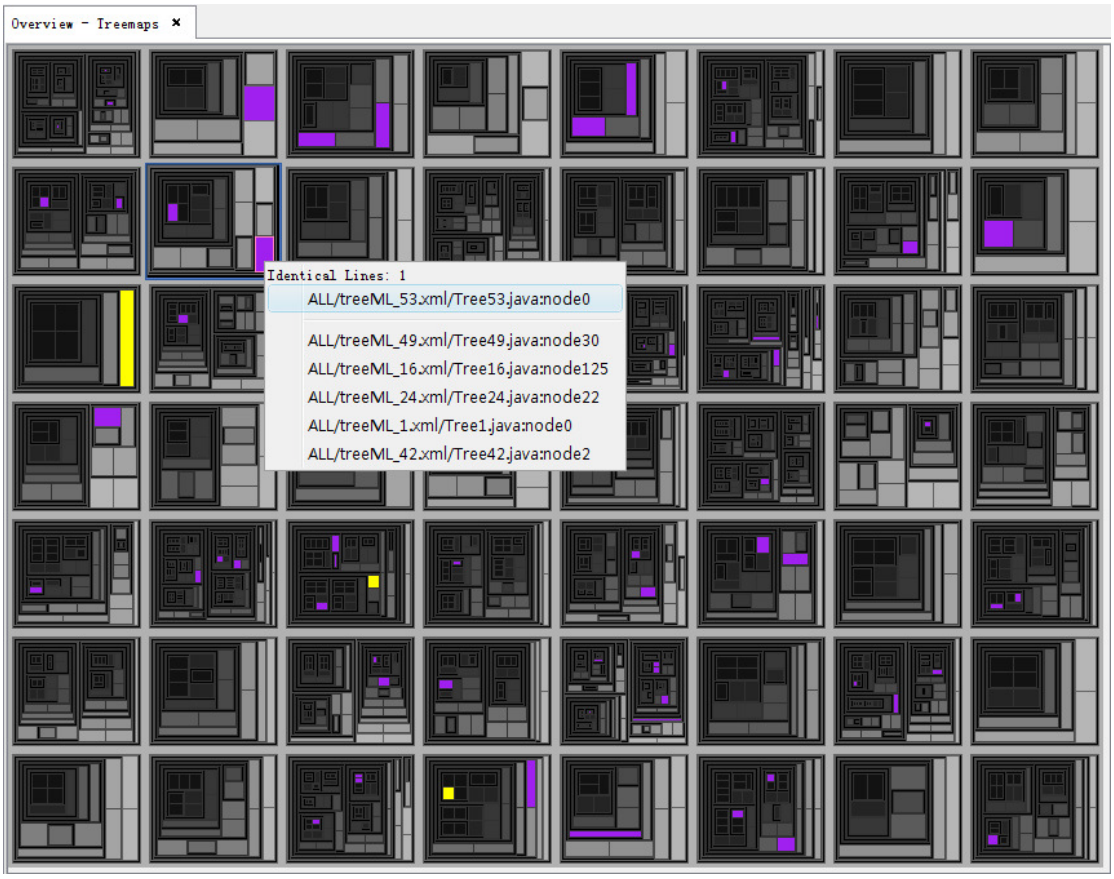

Figure 4.10 Example of showing Popup-Menu (right click)

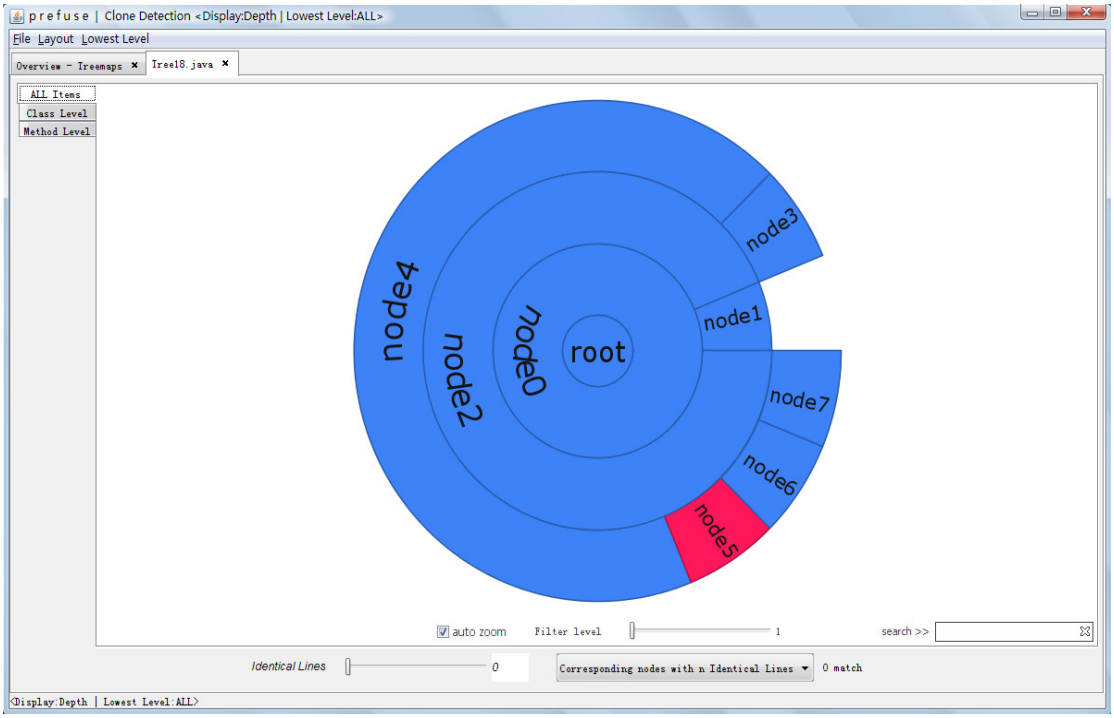

Figure 4.11 Example of radial document visualization (clicked node is red)

In Figure 4.11, as mentioned above, we can see that there are three optional levels

of this radial document visualization at the left side of the screen. They are *ALL Items, Class Level,* and *Method Level.* The user can switch the level of this AST by choosing any level that he or she wants among these three options. The clicked node will still be red in the other two levels, if it is not aggregated into the Class or Method node.

Plus, we can see there are a **Identical Lines slider** and a combo box on the bottom of this screen, if we set the **Identical Lines slider** to a value ***n*** randomly, the nodes which have ***n*** lines in common with currently selected red node will be listed in the combo box aside. And if the value *n* changes, the nodes listed in the Combo Box will change correspondingly.

Selecting a node

To make the user interface more friendly, we add some additional interaction techniques for user to find identical parts with specific **number of identical lines** more easily.

With the key "SHIFT" down, user may select a node that he or she has interests in by left clicking on a node. The selected node turns into **Red**. Meanwhile, **Identical Lines slider** is only available for this selected node at that time. That means if changing the value of **Identical Lines slider** into *n***,** the application will make the nodes, which have more than *n* identical lines in common with the selected Red node, highlighted in Yellow. The visualization will change as the value of **Identical Lines slider** changes. Here is a screen shot showing this technique.

Figure 4.12 Example of showing interaction with selected Node

In this figure, the value of **Identical Lines slider** is 5, so the selected red node has more than 5 lines in common with these yellow nodes. Moreover, with the key "CTRL" down, user can deselect the red node. Once there is no node selected, the **Identical Lines slider** is available again for all nodes.

4.2.3 Additional Interaction Techniques

Changing level of Overview Frame

There is one problem with our treemap interactions in overview frame, that is, only leaf nodes are interactive in those treemaps. That will lead to a problem that Method nodes and Class nodes are invisible to users. To solve this problem, we developed another interaction technique that makes those sub-trees which are rooted by Class node or Method node aggregate into their root node.

As stated above, there are three levels of each AST: *All Items*, *Class Level*, and *Method Level,* which are defined by us in advance. The user can switch levels of all treemaps (ASTs) by clicking the Menu "*Lowest Level*" and choosing any menu item (level) he or she wants. Then Class nodes and Method nodes will be visible. And other interactions like "interaction with top level similarity", "interaction with identical parts" and so on are still the same after transforming the level from one to another. Here are the screen shots of each **Level**. In this Treemap layout, the **area of a node** in treemap depends on the attribute *size* of the node, which means the bigger the size is, the larger area it takes up.

Figure 4.13 Example visualization of *All Items* Level

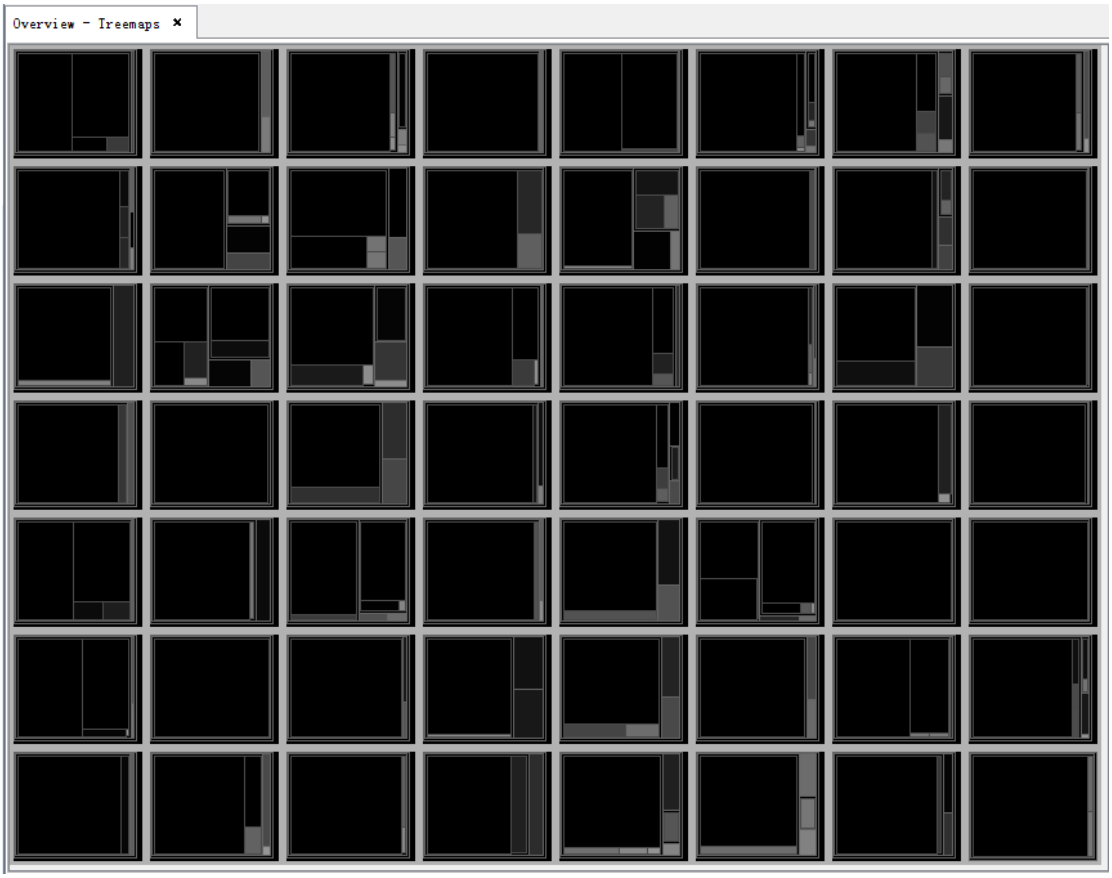

Figure 4.14 Example visualization of Class Level

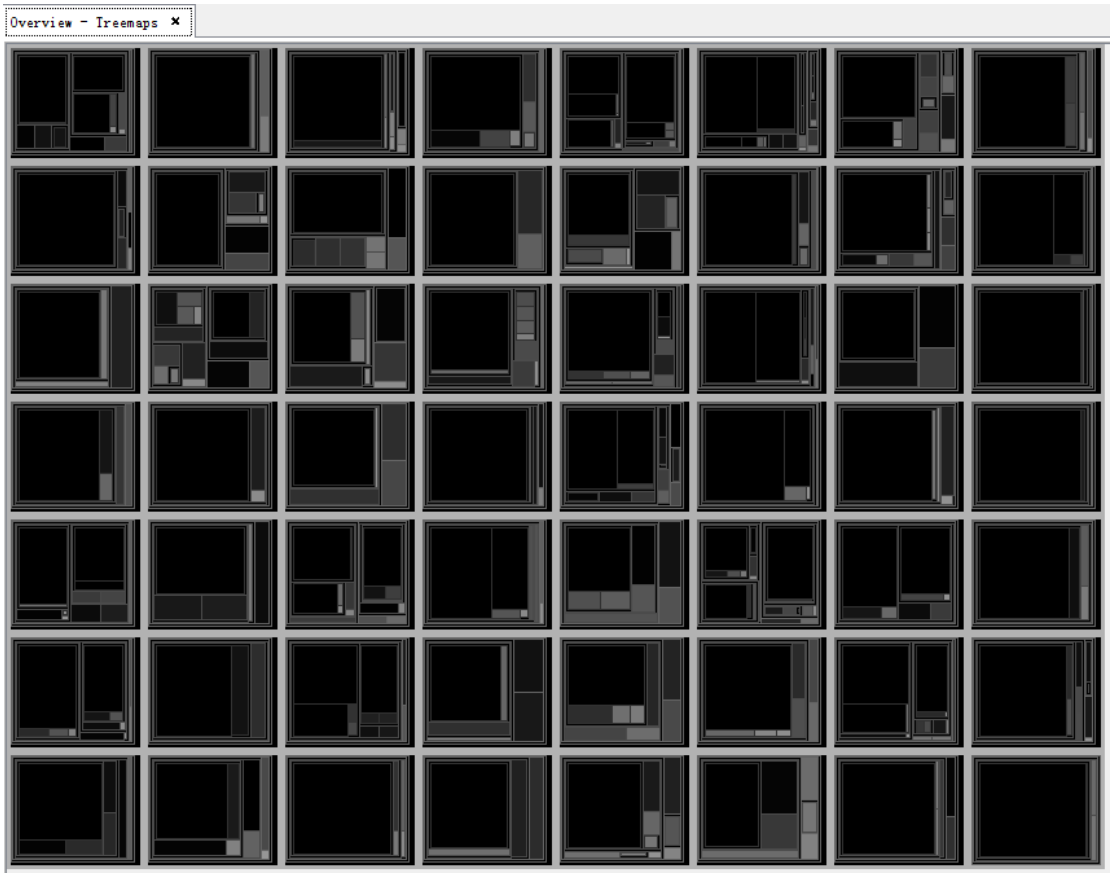

Figure 4.15 Example visualization of Method Level

Keyword and Prefix search

Searching is very helpful when user wants to find the specific contents among those Abstract Syntax Trees. The nodes, the labels of which match user's input, will be highlighted in pink. Moreover, user can check or uncheck the *KeyWord* check box to switch the search model between *Keyword search* model and *Prefix Search* model. The following figure shows the example visualization of *Prefix Search* model. All nodes are highlighted in pink, if the prefix part of their labels match user's input.

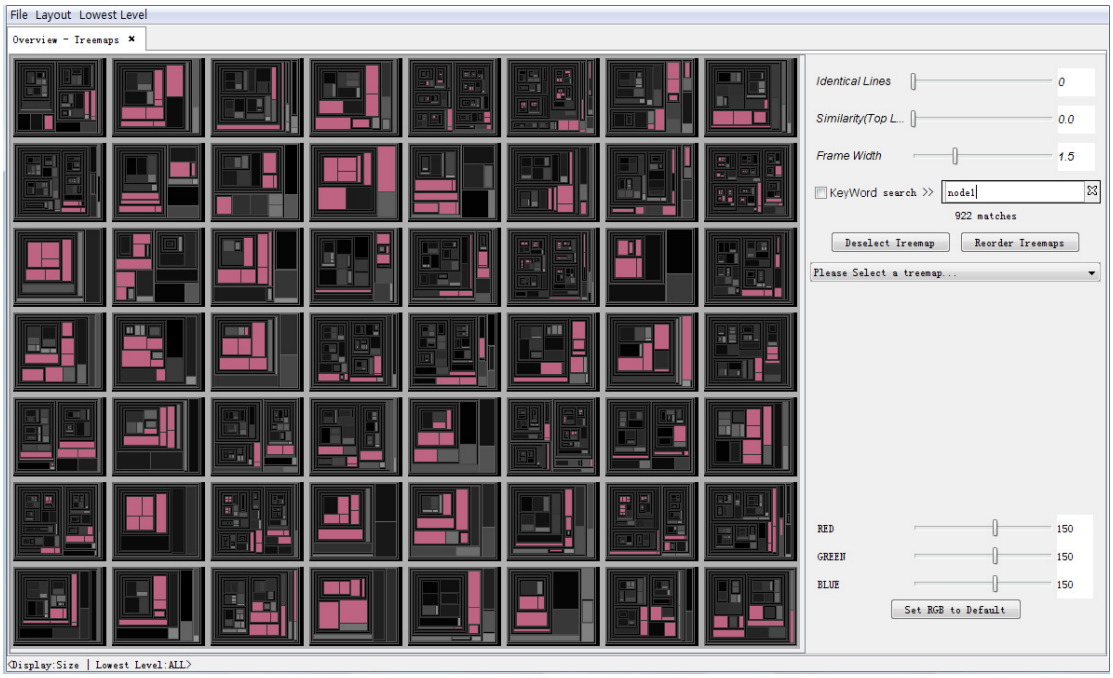

Figure 4.16 Example visualization of *Prefix Search* model

Changing the shade of treemaps

We know that the colors of normal nodes are shaded according to their size or depth in the tree in the beginning. We call this background color here. Sometimes, it is hard for users to see that highlighted nodes with current background color. So we add three sliders for R, G, and B respectively to change the shaded color (Background color) of all treemaps. Moreover, we can change these three sliders to get a clearer contrast of the visualization, or even a clearer hierarchical structure of treemaps. The screen shot of these three sliders is as follows:

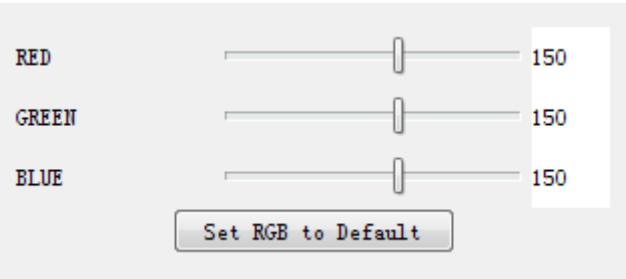

Figure 4.17 Sliders used for changing shaded color of treemaps

Interaction with radial visualization

"DocuBurst" is the first visualization of document content which takes advantage of the human-created structure in lexical databases. The radial visualization of ASTs in our application is based on this "DocuBurst". It has been introduced in detail at [16]. So, we do not have to repeat the interaction techniques and related concepts again. Here is a brief conclusion of its important interaction techniques:

1. Generalized fisheye filter is applied to this visualization, revealing only nodes of depth 1 from focus (root).
2. Search results will become orange. Selected nodes with a single click will become gold.
3. User can select multiple nodes with the help of ctrl-click. And deselect with ctrl-click on a selected node.
4. Graph can be re-rooted by double clicking on any node; new spanning tree is calculated and the graph will be shown centered at that node.
5. If the auto zoom check box is checked, it will reset zoom to fit the data animatedly, when you interacting with it.
6. Expanding or shrinking is also available by dragging the slider down the panel.

With the help of those interaction techniques, we can go through a whole AST in detail without any effort.

5 Implementation

PREFUSE is mainly used to implement this Clone Detection tool and reveal more information. Firstly, we need to do some analysis on the raw data (50-1000 Java files), convert those raw data into the data which can be adapted to our application. So we devise an Interface according to which the users should collect the information (relationship among those Java files) of the raw data. Then this information can be visualized in such a way that can be re-presented by our application. In this chapter, we will first talk about tools that are used for implementation: Grail (Graph Implementation) and PREFUSE. Followed is how we formalize the data that are ready for our tool.

5.1 Grail Library

One of the components with which we are going to develop the visualization tool is Grail graph library that was developed and is being maintained by the School of Mathematics and Systems Engineering of Växjö University. This is a library which is able to create a graph and then do manipulations on it by adding nodes and edges as well as apply graph algorithms to it. Because of availability of its powerful features, Grail can abstract real-world issues so that they can be viewed from a graph perspective. Powerful as Grail is, its structure is not as simple as that of a small-scale project. Also, there are lots of functions in this library and quite a few of them will contribute to our program for this thesis. In the following subsections, important features and functionalities of Grail will be introduced and explained.

5.1.1 A Brief Description of Grail

Grail is a graph library that is being developed at the institution of mathematics and system technology of Växjö University. In general, it is an Application Programming Interface (API) which is able to store and manipulate binary relation between nodes, edges and graphs. Grail's scale is of small-medium size, which contains 11 packages on top level. Each top-level package has 9 classes on average. Every class has 8 methods and 92 lines of codes on average.

5.1.2 The Structure of Grail

Of all the codes in Grail library, there are three packages which are of extreme importance. They are *grail. interfaces*, *grail. iterators* and *grail. properties*. *Grail.interfaces* contains all the interfaces that exist in the system. Though there are 15 of them, the core of Graph part consists of 3 interfaces: *DirectedGraphInterface*, with which we can build a directed graph; *DirectedNodeInterface* with which we can add nodes to a graph; and *DirectedEdgeInterface* which enable us to define the binary relations between nodes.

In the system, a convenient way to traverse all the nodes and edges is by using the Iterator class in Java. There are two kinds of iterators which help a lot when a user wants to check all the nodes or finds whether a node at hand is connected to another

one in the graph. In the library, these two sorts of iterators are noted as *NodeIterator* and *EgdeIterator*. In fact, Grail has its own iterators which have more functions than those in the java library. For example, a node iterator can retrieve the current node it's pointing to by invoking a newly constructed method called "*getNode()*". Other new functions like this really enhance the performance a lot and of course bring about much convenience. Setting an edge iterator to access all the edges is a creative deed. With the iterator, the user can get aware of the relationship of the two nodes (let's imagine them to be two classes in the system we are evaluating) without even visiting them. Especially, when the graph we are looking at is condense, i.e. it has a great quantity of nodes but relatively very few edges, finding two nodes that meet certain requirements is like a mission impossible. However, by checking only a few edges is a piece of cake.

Nodes or edges will not have any meaning if they do not have one or more properties specified. For instance, when we use the Grail system to evaluate a certain piece of software, nodes are probably labeled with properties which signify their identities. In this case, the properties can be "class", "interface" or something. The Grail system has a keyword *type* to present this kind of property with *typeValues* of possible types of entities in the software system. Additionally, another property "name" is expected to be set so that we know which class or interface we are paying attention to. More specific properties can be set to the node if necessary. Besides, attributes of edges are needed in order to help us get a clear idea of the relations between the two nodes connected by it. Back to our evaluation of software example, an edge that has "implements" as its property can explicitly tell the user that node1 which possibly represents an interface has an implementation of itself which identified by node2 provided that the edge stems from node2 and points to node1. In this case, the *LABLE* of the edge would be something like "SetBasedDirectedNode implements DirectedNodeInterface". Figure 5.1 best depicts the internal structure of Grail library.

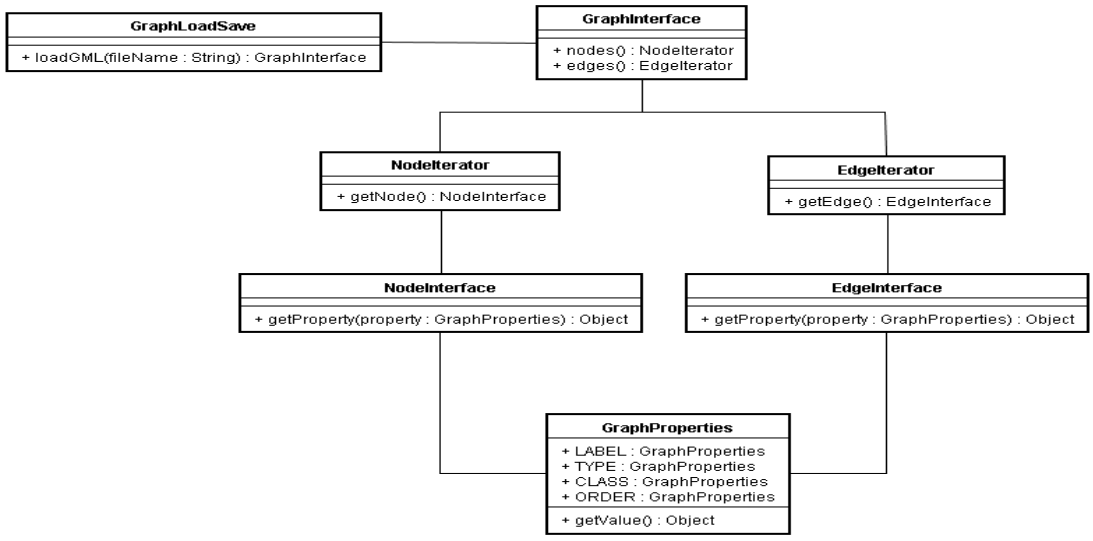

Figure 5.1 The Structure of Grail.

5.1.3 The Benefit of Grail

As introduced above, Grail library has a great capability of processing graphs. This capability includes dealing with tree representation. Since the hierarchies of input Java source files are given by abstract syntax tree, it can be true that using Grail will facilitate our work.

We decide to use Grail library due to the facts below. First, we are very familiar with the functionalities provided by this library. During last year, the two authors of us worked on a thesis which is mainly about extending this capability of this library. The thesis project was to implement some adapters that allow Grail graphs to be converting into different kinds of textual representation for the purpose of saving the graphs on permanent storage. By working with Grail, we got to know this library further and became quite familiar with it. For example, a programming element may be identified as duplicated with another one in the data set. The extent of similarity will be reported as a percentage hence each node in the treemap, no matter leaf or intermediate, may have a property called "similarity". These attributes should be traceable when the data are being visualized. If Grail is used to build a tree representation of the structure of each input document, on one hand, it is rather easy to record every property as well as its corresponding value just by setting a *graphProperty* to the note it belongs. On the other hand, retrieving this value of similarity is hardly effort-consuming. The routine *"getProperty(name, value)"* will return the value of that property called "*name*". In addition, since Grail is developed by a group of experts which are specialized in this area, the reliability of this API is guaranteed.

Another important reason that leads us to make use of this component is that our data set provider happens to be the professor who mainly supervised the development Grail. Professor Löwe from Växjö University is our potential client who provides his analysis result of clone detection among a series of Java source files. Before we can use his data set as the source of the visualization, an interface is necessary so that our procedure can parse the analysis result from him, reads in the information and then converts the texts into graphical elements. Therefore, we had a few meetings with him together with our supervisor Prof. Kerren in order to find a solution. Prof. Löwe said if we use data structures from Grail to store the hierarchical information, he could easily adapt his result to our interface while at that moment the structures of those source files are XML-represented.

Since Grail is so competent with tree processing and our client favors the usage of it, it is wise to take advantage of this library. In the project, Grail is the connector between the original data set and another powerful component Prefuse which is responsible for the visualization. Concrete methods will be covered in the coming chapter that is about design and implementation.

5.2 PREFUSE

Since the major task for this thesis is to *visualize* duplicated parts identified by a so-called anti-cheating machine and the like, another important component that we are

going to use is Prefuse. Prefuse is a very powerful tool kit when dealing with visualization. It is a combination of various procedures that make interactive data visualization possible.

Generally speaking, Prefuse provides a user interface which can be used to develop visualization applications with rich interactive techniques integrated. Apart from various layouts for different types of data and plenty of graphical element encoding methods that involve color, shape, size, Prefuse tends to make offer these wonderful features in a customized way so that the user can combine a set of preferable functions together and then build his own software which is usually even more powerful and target-oriented.

5.2.1 Major Features

The following feature introduction is from Prefuse User's Manual [9]:

- *Table, Graph, and Tree data structures supporting arbitrary data attributes, data indexing, and selection queries, all with an efficient memory footprint.*
- *Components for layout, color, size, and shape encodings, distortion techniques, animation, and more.*
- *A library of interaction controls for common interactive, direct-manipulation operations.*
- *Animation support through a general activity scheduling mechanism.*
- *View transformations supporting panning and zooming, including both geometric and semantic zooming.*
- *Dynamic queries for interactive filtering of data.*
- *Integrated text search using a number of available search engines.*
- *A physical force simulation engine for dynamic layout and animation.*
- *Flexibility for multiple views, including "overview+detail" and "small multiples" displays.*
- *A built in, SQL-like expression language for writing queries to Prefuse data structures and creating derived data fields.*
- *Support for issuing queries to SQL databases and mapping query results into Prefuse data structures.*

5.2.2 Tool Kit Structure

Having a good knowledge of the structure of Prefuse is very helpful for comprehension of how we utilized the functionalities offered in purpose of implementation. Figure 5.2 shows all the packages in the Prefuse tool kit and their relations as well.

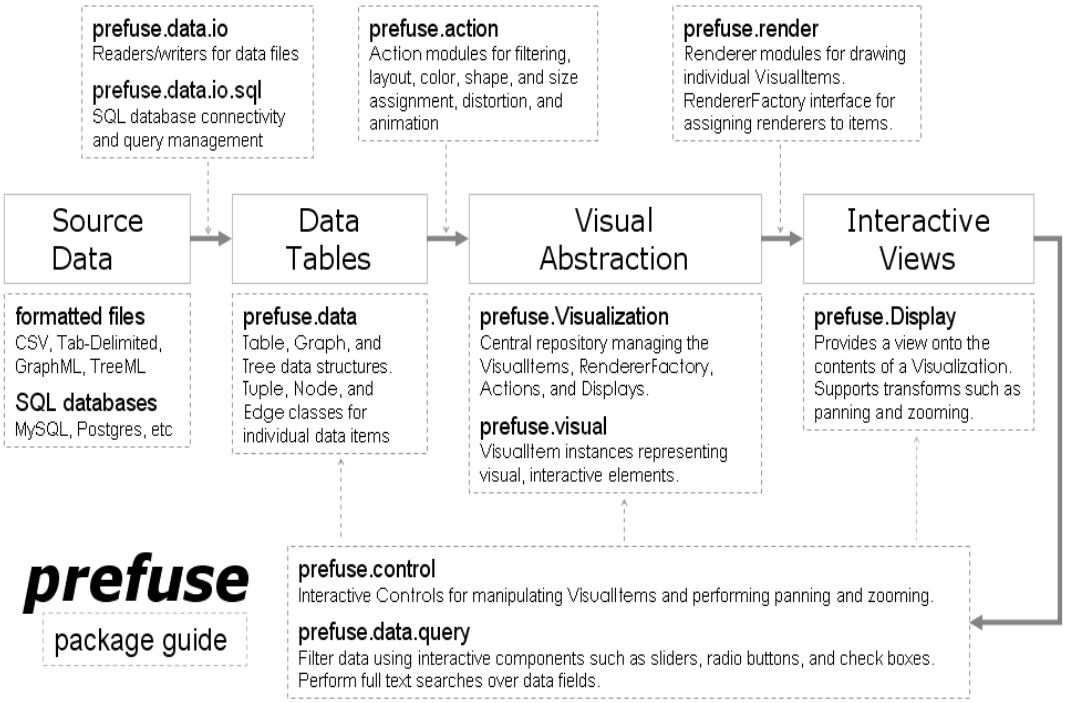

Figure 5.2 Prefuse's Structure, taken from [9]

The data package deals with different data structures that can be used to represent real data. Tables, graphs as well as trees can be transformed so that their information can be interpreted by Prefuse. Prior to visual mapping, proper data tables should be provided as we mentioned in introducing the reference model of information visualization.

As for the data.io package, reading in and writing out information are supported here. The method varies according to the type of the data that we want to read in or output to some external media. However, Prefuse cannot parse any file of any format. Specially-formatted documents should be prepared before Prefuse can read in the information. Take trees as an example, they can only be built if a so-called TreeML file is provided. This type of file is XML-based in which the relations between nodes and their corresponding attributes are specified. When it comes to our thesis project, each root node in those abstract syntax trees will have an attribute called “similarity” which indicates its similarity to other root nodes. The detailed explanation is covered in next chapter where we talk about design and implementation.

5.3 Data Processing

Every Java file has a hierarchy, which can be represented by Abstract Syntax Tree in the form of XML. **Grail** (Graph Implementation) is used to build the data set whose format suits our application. Here comes the specification of Data Set Interface. It helps user understand data set more.

5.3.1 Data Set Interface Specification

The methods need to be implemented are described as follows:

1. `public GraphInterface[] createTrees(int treeNumber);`

This method returns an array of *GraphInterface*s (which are objects of trees). Every Java file can be represented by Abstract Syntax Tree, so this method uses the API offered by Grail to create trees for a number of Java source codes based on their abstract syntax trees. Each tree describes the hierarchy of a Java file. Every node in one tree represents a programming element of Java.

When generating these trees, make sure that the following four attributes have been set for each node:

1) ***Label*** (GraphProperties.LABEL),
2) ***Size*** (GraphProperties.WIDTH), i.e., size of tokens
3) ***Description*** (GraphProperties.DESCRIPTION), i.e.,"isClass","isMethod",or null
   "isClass" means this node represents a Class in the current Java file.
   "isMethod" means this node represents a Method in current Class.
   "null" means this node is neither a class node nor a method node.
4) ***GlobalKey*** (It is unique among all the nodes in different graphs)

2. `public HashMap<Object,DirectedNodeInterface> getKeyToNodeMap();`

This method returns a HashMap that contains the mapping from ***GlobalKey*** to the corresponding ***node*** (*DirectedNodeInterface*). This will be used when developing the visualization and will get the application speed up a little bit.

3. `public HashMap<Double, HashMap<GraphInterface, HashSet<GraphInterface>>> getTopLevelSimilarity(GraphInterface[] trees);`

We need to compare the nodes between different trees to get the similarity values between the documents that contain them, which are called top level similarities.

According to return type of this method, we can easily see that this method returns a ***HashMap(1)*** that contains the mapping from "the value of similarity" to another ***HashMap(2)***.

For example, according to the ***HashMap(1),*** given a specific value of similarity (n%), we can get a ***HashMap(2)*** that containing the mapping from every tree "T" to a set "S" of other Trees. This mapping says that the similarity between each Tree in the set "S" and the tree "T" is "n%" or more, respectively.

4. `public HashMap<Integer, HashMap<DirectedNodeInterface, HashSet<DirectedNodeInterface>>> getIdenticalLinks();`

According to return type of this method, we can easily see that this method returns a ***HashMap(1)*** that contains the mapping from "the number of identical lines" to another ***HashMap(2)***.

The number of identical lines tells that only identical parts that have this number of identical statements with any other files in the data set will be highlighted.

Therefore, a value 1 means that each identical line is important; a value 5 means that only 5 identical lines will be counted.

In reality, maybe the user would like to know that how many identical lines there exist between two class nodes, two method nodes, or other nodes. For example, according to the ***HashMap(1),*** given a specific number "n" of identical lines, we can get a ***HashMap(2)*** that containing the mapping from each node "c" to a set "S" of other nodes says that it contains "n" or more identical lines between each node in the set "S" and the node "c", respectively. The following three methods are similar to this method.

This method can be lower cast into the following three ones. The return types are alike.

1) `public HashMap<Integer, HashMap<DirectedNodeInterface, HashSet<DirectedNodeInterface>>> getIdenticalLinksBetweenClass();`

   In this method, only the nodes which represent class are taken into consideration.

2) `public HashMap<Integer, HashMap<DirectedNodeInterface, HashSet<DirectedNodeInterface>>> getIdenticalLinksBetweenMethods();`

   In this method, only the nodes which represent method are taken into consideration.

3) `public HashMap<Integer, HashMap<DirectedNodeInterface, HashSet<DirectedNodeInterface>>> getIdenticalLinksBetweenNodes();`

   In this method, the nodes apart from class nodes and method nodes are taken into consideration.

5.3.2 Serialize Grail.GraphInterface Object

Once the preprocessed data set is ready, we begin to think about visualizing those data in a proper way. PREFUSE is mainly used to visualize the data set in our project. Since the Object Grail.GraphInterface makes no sense to PREFUSE, we need to serialize the Grail.GraphInterface objects into **XML files** that can be accepted by PREFUSE. Every Grail.GraphInterface object corresponds to one XML file which is so-called XML-based **TreeML** format in PREFUSE.

PREFUSE defined the Schema of XML file for Trees in its own format so-called XML-based TreeML format. It contains a class named *TreeMLReader*. This class is a *GraphReader* instance that reads in tree-structured data in the XML-based TreeML format. TreeML is an XML format originally created for the 2003 Information Visualization conference contest. A DTD (Document Type Definition) for TreeML is available in [10]. Firstly, a simple example is presented as following:

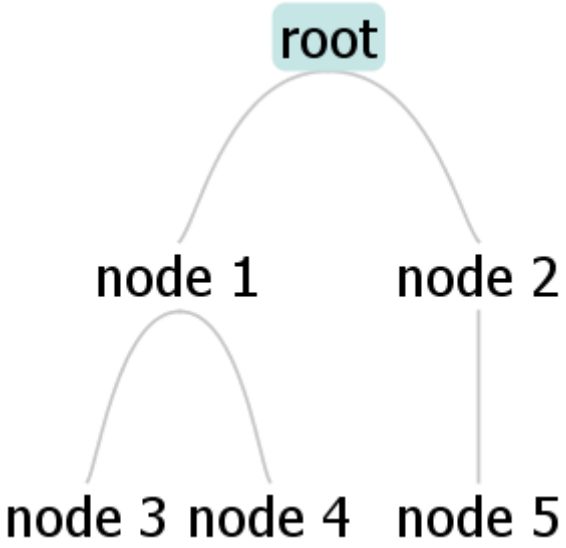

Figure 5.3 A simple Tree

The TreeML-representation corresponds to this figure is:

```
<tree>
  <declarations>
    <attributeDecl name="name" type="String"/>
    <attributeDecl name="size" type="Real"/>
    <attributeDecl name="Description" type="String"/>
    <attributeDecl name="GlobalKey" type="String"/>
  </declarations>
  <branch>
    ……
    <branch>
      ……
    </branch>
    <branch>
      ……
    </branch>
  </branch>
</tree>
```

The beginning part of TreeML "<declarations> …… </declarations>" declares all the attributes that each node contains. We use dom4j [11] to write TreeML files. Please refer to Appendix A.1 for detail of implementation. Here is the main part of the source code:

```
……
//begin of adding declarations
Element element1 = docroot.addElement("declarations");
element1.addElement("attributeDecl").addAttribute("name",
```

```
"name").addAttribute("type", "String");
element1.addElement("attributeDecl").addAttribute("name",
"size").addAttribute("type", "Real");
element1.addElement("attributeDecl").addAttribute("name",
"Description").addAttribute("type", "String");
element1.addElement("attributeDecl").addAttribute("name",
"GlobalKey").addAttribute("type", "String");
//end of adding declarations
……
```

In our application, we need three levels for each Abstract Syntax Tree (Java file). They are named as: “All Items”, “Class Level”, and “Method Level”.

1) ***All Items*:** all the nodes of an abstract syntax tree will be displayed in a Treemap, which means all the nodes will be serialized into TreeML file.
2) ***Class Level*:** each sub-tree which is rooted by a class node will be aggregated into this class node and this new tree will be displayed in a treemap. Only the nodes that are contained in this reduced tree need to be serialized into TreeML file.
3) ***Method Level*:** each sub-tree which is rooted by a method node will be aggregated into this method node and this new tree will be displayed in the treemap. As above, serialize this reduced tree into TreeML file.

When dealing with the data set, all the TreeML files of these three levels are created for every Abstract Syntax Tree at the same time. The Java codes for creating these files are as follows:

```
for (int i = 0; i < treemapNumber; i++) {
model.testWriteXML("ALL",PrefixfileName + (i + 1) + ".xml",trees[i], "");
model.testWriteXML("CLASS",PrefixfileName+ (i + 1) + ".xml",trees[i],
"CLASS");
model.testWriteXML("METHOD",PrefixfileName+ (i + 1) + ".xml",trees[i],
"METHOD");
……
}
```

The TreeML files on different levels are stored into directories named “ALL”, “CLASS”, and “METHOD” respectively.

After this is done, we get all the TreeML files for all Grail.GraphInterface objects (Java files). Then, we can begin our work on visualizing the DATASET, which is one of the most important parts of this project. The visualization approaches have been introduced in previous chapter.

6 Conclusion

In this chapter, we will recall what we have accomplished so far and give a conclusion regarding the tool that has been developed. Also, things to do in the further are included so that the visualization can look even better and the interaction becomes more powerful.

6.1 Achievements

The problem of the thesis was:

Implementing a visualization tool which can visualize the duplicated of Java source files as well as allow accesses to the concrete code. In the hierarchical overview of all the whole data items, powerful interaction techniques are necessary to enable the viewer to check similarities among different documents.

At the beginning, the current situation of information visualization was introduced. Having realized how popular the field is nowadays, some classic examples are mentioned to exemplify the prominent advantages of using graphical elements to illustrate textual messages. When it comes to our thesis project, three convincible motivations are listed out to show the significance of our work here.

Before talking about the developing process of our visualization tool, relevant knowledge is introduced in Chapter 2 to acquaint the readers with the notions and terminologies we used in the following chapters. Firstly, a general explanation about the concept of information visualization is given. Some examples best illustrate the efficiency of this advanced technique compared to text representation of information. In addition, since information visualization is about generating a mental image in the viewer's mind, it is important to know how human perceives things when searching for an approach to visualize things. Therefore, we cover the human perception process prior to the explanation of information visualization reference model. Like every product has to go through a certain process before it can be put on the shelf, three major steps have to be taken so that information visualization is feasible. Data transformation, visual mapping and view transformation consist of the reference model through which we develop our tool. After the basics, an outstanding representation of data relations comes into the sight. Treemap can best describe hierarchical relations, which is exactly what we need for this thesis project. Followed are the introductions of two excellent components that are made sufficient use of at the phase of design and implementation.

Equipped with those useful knowledge and components introduced in Chapter 2, we could finally start to code the serializer/deserializer. First, we planed several steps to complete the serializer/deserializer. During the design, one critical problem was brought out. How could the procedure possibly interpret the data set and read it in. To get the thing done, we designed an interface which connects the two parts together with the help of Grail library. After that, a visual mapping from Grail trees to treemap representation offered by Prefuse was successfully carried out. In addition to that, lots

of interactive techniques followed to help the use with his analysis tasks, like querying and sorting.

So far, this visualization tool has met the requirements both from the very beginning and middle of the development. First of all, given a data set whose structure applies to our interface, a good overview of all the documents' hierarchies can be visualized with treemap representation. The number of read-in files is also flexible to respond to the data sets of different scales. Usually, the program can work on a data set containing 50 to 200 files provided that the memory is sufficiently large. In addition to that, radiant structure is also provided to allow the user to specifically check out the hierarchical detail of a certain document. What's more, concrete codes can show up for inspection in case that the user is interested. These achieved "different levels of abstraction" which is one of the core features of our implementation.

As for the interaction techniques, lots of facilities have been integrated into our tool to help the user with his analysis task. On the overview level, various treemap representations with one feature highlighted are available, such as aggregation of nodes where the granularity of the treemaps' nodes can be altered in purpose of fitting different situations. When it comes to locate the duplicates, nodes can be spotted by moving the "identical lines" slider which can lower or raise the bar of concern. Besides, similarity on the document level can be easily acquired by setting the "similarity" slider to a certain position and the corresponding value indicates the similarity that arouses the user's attention. Apart from that, we even implemented the functionality with which the fore/background colors of those displayed treemaps can be customized, which solved the program of the illegibility when the tool is used on some projectors.

6.2 Future Work

Although our visualization tool is capable of displaying the overview of the whole data set and offers various interactions on the treemaps helping the user with analysis works, there are still several factors that affect the overall performance in a bad way.

First, the learning curve of the tool might be abrupt for those users that have no knowledge of information visualization. Despite that treemaps are very good at revealing relations among data items, our visualization is not so intuitive enough that the user will understand everything at hand when he runs the tool for the first time. Thorough reading through the user manual might be necessary. Second, the appearance of the tool is not as appealing as its functionality is. Making it more visually attractive should be the job to do in the future. Third, the transitory files which are used to generate tree structures with Prefuse are left undeleted on the hard disk when having exited the program. The user may not want them any more after shutdown. Forth, the program is incapable of visualizing duplicated encountered in common text files at the moment. If the structure of a common text file is abstracted into a tree, this program should be able to work properly. These are the things to be done in the future.

6.3 Idea of an Evaluation Design

At the moment of writing, there is no way for us to get a real data set apart from the ones generated randomly by Grail library. However, provided with a concrete data set, we will be able to carry out an evaluation process to see if our project does improve the efficiency of the users with their analysis work.

First of all, there should be a certain number of subjects whom are divided into two groups of the same number of people. In this evaluation process, both of the groups will be asked to do some analysis on a predefined data set. Group A will be provided with all the information in the form of traditional textual representation and Group B can view the data set with our visualization tool. Before starting the analysis, all the subjects are given some time to acquaint themselves with both the format of the textual information and those outstanding interaction techniques offered by the tool.

The analysis work will be carried out by leading the subjects through a questionnaire (the same one for both groups) designed by people who are familiar with the data set. Both groups are supposed to finish all the questions relevant to the data set and at the mean while being timed during the whole process. After all the subjects are done with their questionnaires, the average time spent by each group will be calculated so that we can see which is more efficient. In addition to that, the correctness of the answers will be evaluated, thus we can decide which approach is more effective.

There will also be a second part of this analysis process where the two groups are given the same information but in the other form. This means Group B get the textual data set while the subjects in the other group finally have a chance to try the visualization tool. Another questionnaire of the same complexity as the first one will be handed out to both groups for the analysis work and the average time and the correctness of the answers will be calculated and verified respectively once more.

References

[1] http://www.kottke.org/plus/misc/images/tubegeo.gif. Access date: 2009/05.

[2] Robert Spence, *Information Visualization – Design for Interaction,* 2007.

[3] Andreas Kerren, *Information Visualization I: Course Slide 2*, 2009, Växjö University.

[4] Andreas Kerren, *Information Visualization I: Basics*, 2009, Växjö University.

[5] Stuart K. Card et al, *The perspective wall: detail and context smoothly integrated,* 1991.

[6] Andreas Kerren, *Information Visualization I: Trees*, 2009, Växjö University.

[7] Ying Tu and Han-Wei Shen, *Balloon Focus: a Seamless Multi-Focus+Context Method for Treemaps*, 2008.

[8] Brian Johnson and Ben Shneiderman, *Treemaps: A space-filling approach to the visualization of hierarchical information structures*. In Proc. IEEE Visualization '91, pages 284--291. IEEE CS, 1991.

[9] Source Forge. Prefuse documentation manual. http://prefuse.org/doc/manual/. Access date: 2009/05.

[10] TreeML DTD. http://www.nomencurator.org/InfoVis2003/download/treeml.dtd. Access date: 2009/05.

[11] Source Forge. Dom4j- Introduction. http://www.dom4j.org/dom4j-1.6.1/. Access date: 2009/05.

[12] Bruls, D.M., C. Huizing, and J.J. van Wijk, "Squarified Treemaps" In *Data Visualization 2000, Proceedings of the Joint Eurographics and IEEE TCVG Sumposium on Visualization*, 2000, pp. 33-42.

[13] Ben Shneiderman. Treemap-history. http://www.cs.umd.edu/hcil/treemap-history/. Access date: 2009/05.

[14] John Stasko. SunBurst Page. http://www.cc.gatech.edu/gvu/ii/sunburst/. Access date: 2009/05.

[15] Jeffrey Heer. Information Visualization Projects. Computer Science Department. Stanford University. http://jheer.org/. Access date: 2009/05.

[16] Christopher Collins, Sheelagh Carpendale, and Gerald Penn. *DocuBurst: Visualizing Document Content using Language Structure*. http://kmdi.utoronto.ca/publications/documents/KMDI-TR-2007-1.pdf. Access date: 2009/05.

[17] The University of Alabama at Birmingham. Code Clone Projects - Software Composition and Modeling Laboratory. http://www.cis.uab.edu/softcom/visual/. Access date: 2009/05.

[18] Christopher G. Healey, *Perception in Visualization*, 2009

[19] SeeSoft. A Tool for Visualizing Line Oriented Software Statistics. http://www.cc.gatech.edu/classes/cs7390_98_winter/reports/realsys/seesoft.html. Access date: 2009/05.

[20] Collins, Christopher. *DocuBurst: Document Content Visualization Using Language Structure.* Proceedings of IEEE Symposium on Information Visualization, Poster Session. Baltimore (2006).

[21] Pacific Northwest National Laboratory. IN-SPIRE. http://in-spire.pnl.gov/TipsandTechniques30.pdf. Access date: 2009/05.
[22] The University of Alabama at Birmingham. Clone Detection Literature. http://students.cis.uab.edu/tairasr/clones/literature/. Access date: 2009/05.
[23] Matthias Rieger, Stephane Ducasse. Software Composition Group, University of Berne. *Visual Detection of Duplicated Code.*

Appendix Core Code

A.1 Codes for Creating TreeML files

```
public class WriteToXML {
    final String mlevel = "METHOD";
    final String clevel = "CLASS";
    String currentlevel = "";
    /**
     * de-serialize GraphInterface to TreeML file
     *
     * @param directory
     *     in which directory TreeML file will be stored.
     * @param filePath
     *     name of TreeML file
     * @param tree
     *     GraphInterface(Tree) that will be de-serialized
     * @param lowestLevel
     *     in which lowest Level("ALL","METHOD","CLASS") the
GraphInterface(Tree) will be de-serialized into TreeML file
     * @return
     * @throws Exception
     */
    public String testWriteXML(String directory,String
filePath,GraphInterface tree,
            String lowestLevel) throws Exception {
        currentlevel = lowestLevel;
        DirectedNodeInterface treeroot = this.findroot(tree);
        DefaultTreeView dtv = new DefaultTreeView(
                (DirectedGraphInterface) tree, treeroot);
        dtv.getGraph();
        OutputFormat format = OutputFormat.createPrettyPrint();
        // format.setEncoding("GBK");
        Document document = DocumentHelper.createDocument();
        Element docroot = document.addElement("tree");
        Element element1 = docroot.addElement("declarations");
        element1.addElement("attributeDecl").addAttribute("name",
"name").addAttribute("type", "String");
        element1.addElement("attributeDecl").addAttribute("name",
"size").addAttribute("type", "Real");
        element1.addElement("attributeDecl").addAttribute("name",
"Description").addAttribute("type", "String");
        element1.addElement("attributeDecl").addAttribute("name",
"GlobalKey").addAttribute("type", "String");
        Element root = docroot.addElement("branch");
```

```
        root.addElement("attribute").addAttribute("name",
"name").addAttribute("value", "root");
        Double sizevalue = (Double)
treeroot.getProperty(GraphProperties.WIDTH);
        if(sizevalue==null)
        {
            sizevalue=0.0;
            System.out.println("No size");
        }
        root.addElement("attribute").addAttribute("name",
"size").addAttribute("value", sizevalue.toString());
        String description =
(String)treeroot.getProperty(GraphProperties.DESCRIPTION);
        if(description==null)
        {
            description="null";
        }
        root.addElement("attribute").addAttribute("name",
"Description").addAttribute("value", description);
        String key = (String) treeroot.getKey();
        root.addElement("attribute").addAttribute("name", "GlobalKey")
                .addAttribute("value", key);
        this.traverseChildren(tree, treeroot, dtv, root);
        File f=new File(directory,filePath);
        if(!f.exists())
        {
            f.getParentFile().mkdirs();
            f.createNewFile();
        }
        XMLWriter writer = new XMLWriter(new
FileOutputStream(f.getAbsolutePath()), format);
        writer.write(document);
        writer.close();
        System.out.println("done writing:"+f.getAbsolutePath());
        //return the file path
        return f.getAbsolutePath();
    }
    private void traverseChildren(GraphInterface tree,
            DirectedNodeInterface current, DefaultTreeView dtv, Element
e) {
        NodeIterator childs = dtv.children(current);
        while (childs.hasNext()) {
            childs.next();
            DirectedNodeInterface currentNode = (DirectedNodeInterface)
```

```
childs.getNode();
            Element e1 = null;
            if (currentlevel == mlevel) {
                String nodeType = (String) currentNode
                        .getProperty(GraphProperties.DESCRIPTION);
                if (nodeType == "isMethod" || currentNode.outDegree() ==
0) {
                    e1=this.addAttributeToXML(currentNode, e, "leaf");
                } else {
                    e1=this.addAttributeToXML(currentNode, e, "branch");
                }
                if (nodeType != "isMethod") {
                    this.traverseChildren(tree, currentNode, dtv, e1);
                }
            } else if (currentlevel == clevel) {
                String nodeType = (String) currentNode
                        .getProperty(GraphProperties.DESCRIPTION);
                if (nodeType == "isClass" || currentNode.outDegree() == 0)
{
                    e1=this.addAttributeToXML(currentNode, e, "leaf");
                } else {
                    e1=this.addAttributeToXML(currentNode, e, "branch");
                }
                if (nodeType != "isClass") {
                    this.traverseChildren(tree, currentNode, dtv, e1);
                }
            } else {
                if (currentNode.outDegree() != 0) {
                    e1=this.addAttributeToXML(currentNode, e, "branch");
                } else {
                    e1=this.addAttributeToXML(currentNode, e, "leaf");
                }
                this.traverseChildren(tree, currentNode, dtv, e1);
            }
        }
    }

    private Element addAttributeToXML(DirectedNodeInterface
currentNode,
            Element e, String Type) {
        Element e1 = null;
        e1 = e.addElement(Type);
        String value = (String)
currentNode.getProperty(GraphProperties.LABEL);
```

```
        e1.addElement("attribute").addAttribute("name",
"name").addAttribute("value", value);
        Double sizevalue = (Double) currentNode
                .getProperty(GraphProperties.WIDTH);
        if(sizevalue==null)
        {
            sizevalue=0.0;
            System.out.println("No size");
        }
        e1.addElement("attribute").addAttribute("name",
"size").addAttribute("value", sizevalue.toString());
        String description =
(String)currentNode.getProperty(GraphProperties.DESCRIPTION);
        if(description==null)
        {
            description="null";
        }
        e1.addElement("attribute").addAttribute("name",
"Description").addAttribute("value", description);
        String key = (String) currentNode.getKey();
        e1.addElement("attribute").addAttribute("name", "GlobalKey")
                .addAttribute("value", key);
        return e1;
    }

    private DirectedNodeInterface findroot(GraphInterface dg) {
        NodeIterator nodes = dg.nodes();
        DirectedNodeInterface root = null;
        while (nodes.hasNext()) {
            nodes.next();
            DirectedNodeInterface dnode = (DirectedNodeInterface) nodes
                    .getNode();
            if (dnode.inDegree() == 0) {
                root = dnode;
                break;
            }
        }
        return root;
    }
}
```

A.2 Codes for One Treemap Demo

```
/**
 * Demonstration showcasing a TreeMap layout of a hierarchical data
 * set and the use of dynamic query binding for text search. Animation
 * is used to highlight changing search results.
 *
 */

public class MyTreeMap extends Display {
    private static final long serialVersionUID = 3489888945830439672L;
    // create data description of labels, setting colors, fonts ahead of
time
    private static final Schema LABEL_SCHEMA =
PrefuseLib.getVisualItemSchema();
    static {
        LABEL_SCHEMA.setDefault(VisualItem.INTERACTIVE, false);
        LABEL_SCHEMA.setDefault(VisualItem.TEXTCOLOR,
ColorLib.gray(200));
        LABEL_SCHEMA.setDefault(VisualItem.FONT,
FontLib.getFont("Tahoma",16));
    }
    private boolean isKeywordSearch = true;
    private boolean isSizeDisplayed = true;
    //private boolean isDepthDisplayed = false;
    private static final String tree = "tree";
    private static final String treeNodes = "tree.nodes";
    private static final String treeEdges = "tree.edges";
    private static final String labels = "labels";
    private int displayWidth=700;
    private int displayHeight=600;
    private SearchQueryBinding searchQ;
    private SearchQueryBinding keywordSearchQ;
    private static final String mykeywordsearch = "mykeywordsearch";
    //private static final String myIdenticalNodes="myIdenticalNodes";
    private HashMap<String,Integer> identicalNodesToColor=new
HashMap<String,Integer> ();
    private MySquarifiedTreeMapLayout myTreeMapLayout;
    private StripTreemap stripTreeMapLayout;
    private SliceLayout sliceTreeMapLayout;
    //private Action treemapLayoutAction=null;
    private ColorMap cmapSize = new ColorMap(
        ColorLib.getInterpolatedPalette(100,
            ColorLib.rgb(150,150,150), ColorLib.rgb(0,0,0)), 5, 100);
    private ColorMap cmapDepth = new ColorMap(
```

```
            ColorLib.getInterpolatedPalette(12,
               ColorLib.rgb(255,255,255), ColorLib.rgb(0,0,0)), 0, 12);
    //private ActionList layout;
    Tree tr=null;
    public MyTreeMap(Tree t, String label,String TreeMapLayout) {
        super(new Visualization());
        this.tr=t;
        // add the tree to the visualization
        VisualTree vt = m_vis.addTree(tree, t);
        m_vis.setVisible(treeEdges, null, false);
        // ensure that only leaf nodes are interactive
        Predicate noLeaf =
(Predicate)ExpressionParser.parse("childcount()>0");
        m_vis.setInteractive(treeNodes, noLeaf, false);
        // add labels to the visualization
        // first create a filter to show labels only at top-level nodes
        Predicate labelP =
(Predicate)ExpressionParser.parse("treedepth()=1");
        // now create the labels as decorators of the nodes
       // m_vis.addDecorators(labels, treeNodes, labelP, LABEL_SCHEMA);
        // set up the renderers - one for nodes and one for labels
        DefaultRendererFactory rf = new DefaultRendererFactory();
        rf.add(new InGroupPredicate(treeNodes), new NodeRenderer());
        rf.add(new InGroupPredicate(labels), new LabelRenderer(label));
        m_vis.setRendererFactory(rf);
        // border colors
        final ColorAction borderColor = new BorderColorAction(treeNodes);
        ColorAction fillColor = new FillColorAction(treeNodes);
        // color settings
        ActionList colors = new ActionList();
        colors.add(fillColor);
        colors.add(borderColor);
        m_vis.putAction("colors", colors);
        // animate paint change
        ActionList animatePaint = new ActionList(400);
        animatePaint.add(new ColorAnimator(treeNodes));
        animatePaint.add(new RepaintAction());
        m_vis.putAction("animatePaint", animatePaint);
        myTreeMapLayout=new
MySquarifiedTreeMapLayout(tree,TreeMapLayout);
        myTreeMapLayout.setFrameWidth(1.5);
        stripTreeMapLayout= new StripTreemap(tree);
        //treemapLayoutAction=myTreeMapLayout;
        // create the single filtering and layout action list
```

```
        ActionList layout = new ActionList();
        layout.add(myTreeMapLayout);
        //layout.add(new LabelLayout(labels));
        layout.add(colors);
        layout.add(new RepaintAction());
        m_vis.putAction("layout", layout);
        ActionList striplayout = new ActionList();
        striplayout.add(stripTreeMapLayout);
        //layout.add(new LabelLayout(labels));
        striplayout.add(colors);
        striplayout.add(new RepaintAction());
        m_vis.putAction("striplayout", striplayout);
        sliceTreeMapLayout=new SliceLayout(tree);
        sliceTreeMapLayout.setOrientation(2);
        ActionList slicelayout = new ActionList();
        slicelayout.add(sliceTreeMapLayout);
        //layout.add(new LabelLayout(labels));
        slicelayout.add(colors);
        slicelayout.add(new RepaintAction());
        m_vis.putAction("slicelayout", slicelayout);
        // initialize our display
        setSize(displayWidth,displayHeight);
        setItemSorter(new TreeDepthItemSorter());
       // this.addControlListener(new FocusControl(1));
        this.addControlListener(new DragControl());
        this.addControlListener(new PanControl());
        this.addControlListener(new ZoomControl());
        this.addControlListener(new WheelZoomControl());
        this.addControlListener(new
ZoomToFitControl(Control.MIDDLE_MOUSE_BUTTON));
        this.addControlListener(new NeighborHighlightControl());
        addControlListener(new ControlAdapter() {
            public void itemEntered(VisualItem item, MouseEvent e) {
                item.setStrokeColor(borderColor.getColor(item));
                item.getVisualization().repaint();
            }
            public void itemExited(VisualItem item, MouseEvent e) {
                item.setStrokeColor(item.getEndStrokeColor());
                item.getVisualization().repaint();
            }
        });
        KeywordSearchTupleSet mysearchTS=new KeywordSearchTupleSet();
        keywordSearchQ= new SearchQueryBinding(vt.getNodeTable(),
label,mysearchTS);
```

```
        m_vis.addFocusGroup(mykeywordsearch,
keywordSearchQ.getSearchSet());
        keywordSearchQ.getPredicate().addExpressionListener(new
UpdateListener() {
            public void update(Object src) {
                m_vis.cancel("animatePaint");
                m_vis.run("colors");
                m_vis.run("animatePaint");
            }
        });
        searchQ = new SearchQueryBinding(vt.getNodeTable(), label);
        m_vis.addFocusGroup(Visualization.SEARCH_ITEMS,
searchQ.getSearchSet());
        searchQ.getPredicate().addExpressionListener(new
UpdateListener() {
            public void update(Object src) {
                m_vis.cancel("animatePaint");
                m_vis.run("colors");
                m_vis.run("animatePaint");
            }
        });
        // perform layout
        if(TreeMapLayout.equalsIgnoreCase("slice"))
        {
        m_vis.run("slicelayout");
        }
        else if(TreeMapLayout.equalsIgnoreCase("strip"))
        {
        m_vis.run("striplayout");
        }
        else
        {
        m_vis.run("layout");
        }
    }
    public void changeTreemapLayout(String layoutName)
    {
    if(layoutName.equalsIgnoreCase("strip"))
    {
        m_vis.cancel("layout");
        m_vis.cancel("slicelayout");
        this.invalidate();
        this.reset();
        m_vis.run("striplayout");
```

```
    }
    else if(layoutName.equalsIgnoreCase("Squarified"))
    {
        m_vis.cancel("striplayout");
        m_vis.cancel("slicelayout");
        this.invalidate();
        this.reset();
        m_vis.run("layout");

    }
    else if(layoutName.equalsIgnoreCase("Slice"))
    {
        m_vis.cancel("layout");
        m_vis.cancel("striplayout");
        this.invalidate();
        this.reset();
        m_vis.run("slicelayout");
    }
    }


    /**
     * indicates if size of node will be reflected by the area of the node
or not
     * @param size
     *      size will be reflected by the area of the node (true), otherwise,
false
     *
     */
    public void setSizeConsidered(boolean size)
    {
    myTreeMapLayout.setSizeVisible(size);
    myTreeMapLayout.setDepthVisible(!size);
    this.invalidate();
        this.reset();
    m_vis.run("layout");
    }

    public void setToDefaultLayout()
    {
    myTreeMapLayout.setSizeVisible(false);
    myTreeMapLayout.setDepthVisible(false);
    this.invalidate();
        this.reset();
```

```
    m_vis.run("layout");
    }

    public void setDepthVisible( boolean depth)
    {
    myTreeMapLayout.setDepthVisible(depth);
    myTreeMapLayout.setSizeVisible(!depth);
    this.invalidate();
        this.reset();
    m_vis.run("layout");
    }

    public void SetTreemapWidth(int width)
    {
    displayWidth=width;
    }

    public void SetTreemapHeight(int width)
    {
    displayHeight=width;
    }

    public SearchQueryBinding getSearchQuery() {
        return searchQ;
    }

    public SearchQueryBinding getKeyWordSearchQuery() {
        return keywordSearchQ;
    }

    /**
     * indicates if this treemap currently supports keyword search or not
     * @param b
     *  keyword search is active (true), prefix search model is
active(false)
     */
    public void setKeywordSearch(boolean b) {
         isKeywordSearch=b;
         m_vis.cancel("animatePaint");
        m_vis.run("colors");
        m_vis.run("animatePaint");
       }
   /**
    * @param b
```

```
* normal nodes are shaded according to their
* size of node(true). otherwise,
* they are shaded according to their depth in the tree(false)
*/
public void setSizeVisible(boolean b) {
    isSizeDisplayed=b;
     m_vis.cancel("animatePaint");
    m_vis.run("colors");
    m_vis.run("animatePaint");

   }
/**
 * specified node with the key "nodekey" is highlighted in color "color"
 *
 * @param nodekey
 *       the node with this node key will be highlighted
 * @param color
 *       highlighted color
 */
public void setHighLighted(String nodekey,int color)
{
identicalNodesToColor.put(nodekey, color);
m_vis.cancel("animatePaint");
    m_vis.run("colors");
    m_vis.run("animatePaint");

}

public void clearIdenticalNodes()
{
identicalNodesToColor.clear();
m_vis.cancel("animatePaint");
   m_vis.run("colors");
   m_vis.run("animatePaint");
}
public int getNodeColor(String key)
{
return identicalNodesToColor.get(key);
}
/**
 * the color map for depth is changed according to r,g,b
 * @param r
 * @param g
 * @param b
```

```
 */
public void changeDepthColorMap(int r,int g,int b)
{
cmapDepth = new ColorMap(
          ColorLib.getInterpolatedPalette(12,
              ColorLib.rgb(r,g,b), ColorLib.rgb(0,0,0)), 0, 12);
m_vis.cancel("animatePaint");
    m_vis.run("colors");
    m_vis.run("animatePaint");
}

/**
 * the color map for size is changed according to r,g,b
 * @param r
 * @param g
 * @param b
 */
public void changeSizeColorMap(int r,int g,int b)
{
cmapSize = new ColorMap(
       ColorLib.getInterpolatedPalette(100,
           ColorLib.rgb(r,g,b), ColorLib.rgb(0,0,0)), 5, 100);
m_vis.cancel("animatePaint");
    m_vis.run("colors");
    m_vis.run("animatePaint");
}

public void setFrameWidth(double w) {
 myTreeMapLayout.setFrameWidth(w);
// this.invalidate();
     this.reset();
     this.removeAll();
 m_vis.run("layout");
}

/**
 * Set the stroke color for drawing treemap node outlines. A graded
 * grayscale ramp is used, with higher nodes in the tree drawn in
 * lighter shades of gray.
 */
public static class BorderColorAction extends ColorAction {
    public BorderColorAction(String group) {
        super(group, VisualItem.STROKECOLOR);
    }
```

```
        public int getColor(VisualItem item) {
            NodeItem nitem = (NodeItem)item;
            if ( nitem.isHover() )
                return ColorLib.rgb(255,130,191);
            int depth = nitem.getDepth();
            if ( depth < 2 ) {
                return ColorLib.gray(100);
            } else if ( depth < 4 ) {
                return ColorLib.gray(75);
            } else {
                return ColorLib.gray(50);
            }
        }
    }

    /**
     * Set fill colors for treemap nodes. Search items are colored
     * in pink, while normal nodes are shaded according to their
     * size(or depth) in the tree.
     */
    public class FillColorAction extends ColorAction {
        public FillColorAction(String group) {
            super(group, VisualItem.FILLCOLOR);
        }

        public int getColor(VisualItem item) {
            if ( item instanceof NodeItem ) {
                NodeItem nitem = (NodeItem)item;
                if ( nitem.getChildCount() > 0 ) {
                    return 0; // no fill for parent nodes
                } else {
                 if(isKeywordSearch&&m_vis.isInGroup(item,
mykeywordsearch))
                 {
                     return ColorLib.rgb(191,99,130);
                 }
                 else if(!isKeywordSearch&& m_vis.isInGroup(item,
Visualization.SEARCH_ITEMS))
                 {
                     return ColorLib.rgb(191,99,130);
                 }
                 else
if(identicalNodesToColor.containsKey(nitem.getString("GlobalKey")))
```

```
                {
                    return
identicalNodesToColor.get(nitem.getString("GlobalKey"));
                }
                else
                {

                    if(isSizeDisplayed)
                    {
                        return
cmapSize.getColor(nitem.getDouble("size"));
                    }
                    else
                    {
                        return cmapDepth.getColor(nitem.getDepth());
                    }
                }
               }
            } else {
                return cmapDepth.getColor(0);
            }
        }
    } // end of inner class TreeMapColorAction

    /**
     * Set label positions. Labels are assumed to be DecoratorItem
instances,
     * decorating their respective nodes. The layout simply gets the bounds
     * of the decorated node and assigns the label coordinates to the center
     * of those bounds.
     */
    public static class LabelLayout extends Layout {
        public LabelLayout(String group) {
            super(group);
        }
        public void run(double frac) {
            Iterator iter = m_vis.items(m_group);
            while ( iter.hasNext() ) {
                DecoratorItem item = (DecoratorItem)iter.next();
                VisualItem node = item.getDecoratedItem();
                Rectangle2D bounds = node.getBounds();
                setX(item, null, bounds.getCenterX());
                setY(item, null, bounds.getCenterY());
            }
```

```
        }
    } // end of inner class LabelLayout

    /**
     * A renderer for treemap nodes. Draws simple rectangles, but defers
     * the bounds management to the layout.
     */
    public static class NodeRenderer extends AbstractShapeRenderer {
        private Rectangle2D m_bounds = new Rectangle2D.Double();

        public NodeRenderer() {
            m_manageBounds = false;
        }
        protected Shape getRawShape(VisualItem item) {
            m_bounds.setRect(item.getBounds());
            item.getSize();
           // m_bounds.setRect(item.getX(), item.getY(), w, h);
            return m_bounds;
        }
    } // end of inner class NodeRenderer
} // end of class TreeMap
```

A.3 Codes for Main Frame

```
    /**
     *  All treemaps, Sliders, button and search box are displayed in this
panel
     *
     * @param label
     * @return
     *      panel
     */
    public JComponent mainPanel(String label) {

        // similarity Slider
        JValueSlider similaritySlider = new JValueSlider(
                "Similarity(Top Level)", 0.0, 1.0, 0.0);
        similaritySlider.setPreferredSize(new Dimension(300, 30));
        similaritySlider.setMaximumSize(new Dimension(300, 30));
        similaritySlider.setFont(new Font("", Font.ITALIC, 12));
        similaritySlider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                Double similarity = ((JValueSlider)
e.getSource()).getValue().doubleValue();
                currentSimilarity = similarity;
```

```
                int i=GcomBox.getSelectedIndex();
                if(i>0)
                {
                    showSimilarTreemapsToSelected();
                }
                else if(i==0)
                {
                    System.out.println(similarity);
                    showSimilarTree(similarity);
                    treeBorderRepaint();
                }
            }
        });
        // identical detection slider
       JValueSlider identicalSlider = new JValueSlider("Identical Lines",
0,
                10, 0);
        identicalSlider.setPreferredSize(new Dimension(300, 30));
        identicalSlider.setMaximumSize(new Dimension(300, 30));
        identicalSlider.setFont(new Font("", Font.ITALIC, 12));
        identicalSlider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                currentIdenticalLines = ((JValueSlider)
e.getSource()).getValue()
                .intValue();
                if(isAnItemSelected)
                {

    showIdenticalNodesWithSelected(currentIdenticalLines);
                }
                else
                {
                    treemapRepaint();
                    showIdenticalNodes(currentIdenticalLines);
                }
            }
        });
        // identical detection slider
        JValueSlider frameSlider = new JValueSlider("Frame Width", 0.0,
                5.0, 1.5);
        frameSlider.setPreferredSize(new Dimension(300, 30));
        frameSlider.setMaximumSize(new Dimension(300, 30));
        frameSlider.setFont(new Font("", Font.ITALIC, 12));
        frameSlider.addChangeListener(new ChangeListener() {
```

```
            public void stateChanged(ChangeEvent e) {


    if(SquarifiedTreeMapLayout.isSelected()||sizeTreeMapLayout.isSele
cted())
                {
                    double framewidth = ((JValueSlider)
e.getSource()).getValue().doubleValue();
                    for (int i = 0; i < treemapArray.length; i++) {
                        MyTreeMap treemap = (MyTreeMap)
treemapArray[i].getComponent(0);
                        treemap.setFrameWidth(framewidth);
                    }
                }
            }
        });

        JMySearchPanel jsp = new JMySearchPanel(treemapArray, true);

        JButton orderButton=new JButton("Reorder Treemaps");
        orderButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                reOrderTreemaps();
            }
        });
        JButton deSelectButton=new JButton("Deselect Treemap");
        deSelectButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                GcomBox.setSelectedIndex(0);


            }
        });
        JScrollPane sp = new JScrollPane(treemapArea);
        GcomBox=new JComboBox();
        box = new Box(BoxLayout.Y_AXIS);
        jsp.setShowKeywordCheckBox(true);
        jsp.setShowResultCount(true);
        box.add(Box.createVerticalGlue());
        box.add(Box.createVerticalStrut(10));
        box.add(identicalSlider);
        box.add(Box.createVerticalStrut(10));
        box.add(similaritySlider);
        box.add(Box.createVerticalStrut(10));
        box.add(frameSlider);
```

```
        box.add(Box.createVerticalStrut(10));
        box.add(jsp);
        box.add(Box.createVerticalStrut(10));
        Box buttonBox = new Box(BoxLayout.X_AXIS);
        buttonBox.add(deSelectButton);
        buttonBox.add(Box.createHorizontalStrut(10));
        buttonBox.add(orderButton);
        box.add(buttonBox);
        box.add(Box.createVerticalStrut(10));
        box.add(GcomBox);
        box.add(Box.createVerticalStrut(250));
        rgb=sizergb;
        box.add(rgb);
        orderButton.setToolTipText("Re-order highlighted tree-maps
according to the number of files that are similar with each file in current
similarity");
        JPanel rightPanel = new JPanel();
        rightPanel.add(box,BorderLayout.NORTH);
        GcomBox.addItem("Please Select a treemap...");
        GcomBox.setEditable(false);
        GcomBox.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent arg0) {
                doComBoxSelection();
                if(GcomBox.getSelectedIndex()!=0)
                    showSimilarTreemapsToSelected();
                else
                    showSimilarTree(currentSimilarity);
            }
        });
        for(int i = 0; i < treemapNumber; i++)
        {
            GcomBox.addItem(trees[i]);
        }
        JPanel contentPanel = new JPanel();
        contentPanel.setLayout(new BorderLayout());
        contentPanel.add(sp, BorderLayout.CENTER);
        contentPanel.add(rightPanel, BorderLayout.EAST);
        return contentPanel;
    }
```

Växjö
University

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

Tel. +46 (0)470 70 80 00, fax +46 (0)470 840 04
http://www.vxu.se/msi/