MASTER’S THESIS

Panoramix,
a panoramic visualisation tool for metrics

by
L.B.K. van Daal

Supervisor:  dr. C. Huizing

Eindhoven, October 17, 2005
Abstract

Large software systems are hard to grasp. To aid the understanding of such systems, software metrics have been used for a fairly long time. A large Object Oriented system with a metric value for each element, results in an almost unmanageable amount of values for each metric used to measure the system. The Panoramix visualisation tool gives a panoramic view of the entire system, and on top of this, metrics are visualised. This panoramic view gives the user the ability to view the entire system in one glance without imposing a high level structure view of the system. Viewing metrics on top of this structure provides a way for the user to find elements of interest, and recognise patterns, which in turn should aid code handling and modifying tasks.
# Contents

1 Introduction .............................................. 5

2 Software Visualisation ..................................... 6

3 Visualisation structure .................................... 10
   3.1 Overview layout ................................... 10
   3.2 Layout problem ................................... 11
   3.3 Class relation view layout ......................... 13

4 Visualised data .......................................... 15
   4.1 Metrics ........................................... 15
      4.1.1 Basic code complexity metrics ............... 15
      4.1.2 Informational complexity metrics ............ 16
      4.1.3 Design metrics ................................ 17
   4.2 Relations .......................................... 17
      4.2.1 Call relation ................................ 18
      4.2.2 Inheritance relation ......................... 18
      4.2.3 Access relation .............................. 18
      4.2.4 Use relation ................................ 18

5 Metric visualisation .................................... 19
   5.1 Colour coding ..................................... 19
   5.2 Multiple display dimensions ....................... 21
   5.3 Preprocessing ..................................... 22
   5.4 Conclusion ........................................ 22

6 Application of Panoramix ............................... 23
   6.1 Complexity measure ................................ 23
   6.2 Relations ......................................... 26
   6.3 Patterns .......................................... 27

7 Final remarks ........................................... 32
   7.1 Conclusion ........................................ 32
   7.2 Further work ...................................... 32

A Example source code ................................... 34
   A.1 escape ........................................... 34
   A.2 group0 ............................................ 37
Chapter 1

Introduction

There are currently many large software systems, all of which are fairly complex and are developed by teams of programmers. Once a project is past its general design phase, it becomes increasingly more difficult to keep an overview of the entire system. It is also not uncommon that people new to an existing system need to gain insight into it. It is very hard to transfer experience and knowledge to people new to a system, if people with experience are even available. To improve the knowledge someone has of a certain system it is almost impossible to read and understand all the source code of the system. The next best thing would be to look only at interesting elements. To this end there are quantitative measures for source code, called metrics.

Given a large project consisting of many elements, analysing the list of metrics for each element is still a complex task. It is furthermore very hard to discern patterns between metrics or between elements of the software system. To aid comprehension it is possible to visualise the available data, which is exactly the aim of this project. More specifically, the aim is to develop a visualisation structure of Java code in a medium to large project without imposing a too specific view on the user. Using this visualisation of the source code as a base, metrics pertaining to this source code can be visualised. This is meant to give an insight into the code base. In statistics this type of data analysis is called Exploratory Data Analysis (EDA) [Kotz et al., 1982]. The final program uses external analysis programs to obtain metrics, see section 4.1 for descriptions of these metrics.

The choice for a static panoramic visualisation was taken to keep the users working memory as free as possible to grasp the structure of the system, and of the metrics pertaining to that system. If the view where not static, then the user would have to interact with the system to mentally get an overview of the system, this task would be done in the visuospatial sketchpad, which is part of the working memory. This part is used to manipulate and temporarily store visual and spatial information, according to the model of working memory by Baddeley [Baddeley, 1986]. Which element(s) of this model would be taxed by a task such as grasping a complex software system is not easily deduced. But whichever element that would be, it is certain that the central executive, which is the cognitive process that manages the working memory, would be taxed more heavily because the visualisation is not a static panoramic one. This increased workload would have a negative effect on the users ability to comprehend the complex software system. Baddeley refers to dysexecutive syndrome (DES) as an example of incorrect functioning of the central executive, which results in “a difficultly in grasping the whole of a complicated state of affairs”.

Chapter 2 provides an overview of the area of Software Visualisation. After that, chapter 3 shows the structure chosen for the visualisation of the source code. Chapter 4 gives more insight into the type of information that is visualised, and chapter 5 handles the way in which this data, specifically software metrics, can be visualised. After that, chapter 6 shows an example use case. And finally chapter 7 gives some final remarks and further work.
Chapter 2
Software Visualisation

A project such as this falls under the heading of Software Visualisation (SV). Over the years many programs have been developed to aid understanding of programs and algorithms. This section is an overview of existing software visualisations. It is by no means a comprehensive list. The aim was to provide an overview of the field of SV as it currently stands. For a taxonomy of SV we defer to an article by Price et al. [Price et al., 1993], which gives a good overview of the areas of interest in SV.

Software visualisation can be divided into two parts: Algorithm Visualisation (AV) and Program Visualisation (PV). AV is used to show and possibly design the workings of an abstract algorithm, while PV shows an aspect of an existing program. There are several aspects of a program that can be visualised, examples are: source code, runtime behaviour, and changes to the code. PV can furthermore be split up into the areas of Static Software Visualisation (SSV) and Dynamic Software Visualisation (DSV). A program that falls in the former category shows the software based on the source code, while a program from the latter category shows a system based on the run-time behaviour. Panoramix displays software metrics on a structure representing the source code, so it falls in the category of SSV. The following text describes some programs that are related to Panoramix in some way.

An example of AV is the rube framework [Hopkins and Fishwick, 2003], which allows showing the working of a program using an appropriate metaphor. The example given in the paper is that of a task scheduler as a part of an Operating System (OS), where the OS is shown as a building with people inside it representing the tasks. This example comes from an existing system, which would seem to place this program in the category DSV. The rube system, however is an AV, since the behaviour of the task scheduler has to be abstracted, by a programmer, to a system that can be visualised. This program is more suited to teaching the workings of a certain program or algorithm, since there has to be someone who already understands the workings and has to translate them to a system that can be visualised by the rube framework.

Relvis [Pinzger et al., 2005] is an examples of an SSV, the first of five such examples. It shows the evolution of software through time. In particular it shows, for each module, a collection of values indicating some aspect of the source code. Certain relations between the modules are also shown by rectangles connecting two modules, where the width and length of the rectangles display two values that apply to the connection between the two modules. The metrics for the modules are visualised through a Kiviat Diagram. This is a diagram with multiple axes, along each axis a value in a unique domain is a point on the axis. By connecting these points a polygon is shown that is unique for the state that the module is in (according to the metrics). The graph formed by connecting the modules represented by Kiviat Diagrams is called a Kiviat graph. An example of such a graph is shown in figure 2.1.

The Visual Code Navigator (VCN) [Lommerse et al., 2005], is a set of three tools that visualise source code in some way. Each tool defines a view on the code. The syntactic view shows the source code with extra formatting to show the syntactic constructs. A second view
Figure 2.1: Kiviat Graph of 7 Mozilla modules implementing the functionality for handling the content and layout of web-sites. Each diagram presents 20 different source code and evolution metrics of software modules of 7 subsequent releases. Edges indicate coupling dependencies between the modules. Adapted from [Pinzger et al., 2005].

shows the objects that are produced from a source code file during compilation, this part is called the symbol view. Finally, the evolution view shows the change to a number of source files through the lifetime of a project. The part of the VCN that is most closely related to Panoramix is the symbol view. This tool shows, among other things, classes, functions and fields in a cushioned squarified treemap (see figure 2.2). Relationships between those elements are shown by highlighting them and connecting them with arrows.

Balzer et al. [Balzer et al., 2004] have created a landscape visualisation of a software system. The term landscape is used loosely here, since the resulting visualisation is relatively abstract (see figure 2.3). The packages of a certain system are depicted as spheres, within these spheres can be other spheres or a flat square. These squares contain circular areas which represent the classes, these areas are cities with buildings of different types contained in them to indicate the presence of methods and fields. This structure shows the OO hierarchy of a certain system. Using this hierarchy, relations between the elements of the system can be shown (for example inheritance).

More related to Panoramix is Source Viewer 3D (sv3D) [Marcus et al., 2003]. It gives a three dimensional representation of source code, see figure 2.4. The program uses external tools to retrieve information about the code. It displays the information based on a structure that is derived from the lines of code, each line of code is represented by a 3D structure that can display several metrics concurrently. The 3D structure used is a bar, which can vary in height, position, and colour. A possible extension of this framework to “allow definition of mappings that will represent the software system at higher abstraction levels” is mentioned as future work in the article.
Figure 2.2: The symbol view made by the Visual Code Navigator \cite{Lommerse2005}

Figure 2.3: A software landscape of 'SystemX', adapted from Balzer et al. \cite{Balzer2004}

Treemaps, already mentioned in context with The Visual Code Navigator \cite{Lommerse2005}, are generally made using squares. An adaptation to this is the Voronoi Treemap \cite{Balzer2005}, which subdivides an area into polygons. As with squarified treemaps, Voronoi treemaps can also be used to visualise a source code structure. The size of the polygons can then relate information about a certain metric, as can the colour. Transparency can be used to layer several objects on top of each other to convey more hierarchy information (see figure 2.5). The Voronoi treemap uses a diagrammatic structure to display an entire software system, which is similar to the aim of Panoramix.
Figure 2.4: Elements of a visualisation made by sv3D. Adapted from Marcus et al., 2003

Figure 2.5: A Voronoi Treemap of ‘JFree’ Adapted from Balzer et al., 2005
Chapter 3

Visualisation structure

3.1 Overview layout

The number of ways to visualise the Java code of a project may seem infinite. A good visualisation has to adhere to certain rules however. One of these is the principle of *pictorial realism* [Roscoe, 1968]. This principle dictates that the visualisation should look like the entity it represents. Since the entity that we are visualising is always an Object Oriented (OO) program it seems obvious to use this structure in a visualisation. This OO structure is basically an acyclic graph. The root node is the entire program, which has the Java packages as child nodes. These packages, in turn, can have classes or other packages as child nodes. The classes can contain other classes, methods and fields. This tree is simplified by elevating all the packages to the same level. The same is done with classes. When this is done, the nested packages and nested classes have the same depth in the tree. This simplification allows for a more efficient use of screen space. The need for a clear and simple visualisation is considered more important for this project than showing the hierarchy in greater detail.

![Class Package Class Package](image)

Figure 3.1: The layout

Giving each element a rectangle in which it can be drawn creates a versatile yet visually simple system. To show the hierarchy, the elements are stacked such that the package is on top and the methods and fields on the bottom. A limit to the height of methods and fields is put in place and the methods and fields are placed next to each other such that this limit is not exceeded (see Figure 3.1). The methods and fields are laid out, in reading order, in the same sequence as they occur in the source code, to preserve consistency. To make good use of the available space the display window is divided into horizontal bands, called lines. These lines form one single rectangle if each one is connected on the left side to the one above it, and on the right side to the one below it. The problem of finding an efficient layout for a given tree is not trivial, and will be handled in the next section.

Now that we have a layout to use, we can start devising forms to fit inside the layout. Since the layout was created to be visually simple, we also wish the forms that fit inside the
layout to be simple. Again, the simplest choice is rectangles that fit nicely in the alloted rectangular area. To indicate broken packages a notch and similarly shaped hole are used. Since all method and field elements of a class are grouped together under their respective class element it is hard to distinguish the difference. This would not be a big problem if methods and fields where relatively similar elements, but they are not. To show this difference we use a square for a method, and a square with a missing corner for a field. The resulting layout can be seen in figure 3.2. On top of this layout each package is labelled with its name, to aid orientation.

### 3.2 Layout problem

To fit all the elements on the screen as described above leads to certain problems, which lead to wasted space $w$. The first problem is the scaling problem, which causes wasted space $w_s$. This problem occurs because the methods and fields are kept square, and the maximum number of methods and fields that can fit on a line vertically is kept constant. Say $s_h$ and $s_w$ are the height and width in pixels of the screen, and $n$ is the number of lines in which the screen is divided (see figure 3.4). Say furthermore that $e_w$ and $e_h$ is the width and height in blocks of all the elements laid out next to each other in the formation described in the previous section, here one block is the size of one method or field square. Since the methods and fields are square and there is a maximum number of methods and fields that fit vertically, the aspect ratio of $e_w : e_h$ is not variable. The width and height of the single line in which all the elements should fit are $s_w \cdot n$ and $s_h / n$. The only variable available to modify the aspect ratio of the line to fit all the elements is $n$, which is a natural number. The optimal value for $n$ would be $n_o = \sqrt{(e_w \cdot s_h) / (e_h \cdot s_w)}$. If this $n_o$ is rounded to the nearest integer number then we can say that $n = n_o + e$ for $-0.5 \leq e \leq 0.5$. More specifically if $n_o$ is rounded up $(0 < e \leq 0.5)$ then the single line is too wide, which means that the elements can be scaled up such that they will fit in the height and the wasted space will be due to the extra width (see figure 3.3). Conversely if $n_o$ is rounded down ($-0.5 \leq e < 0$) then space will be wasted due to extra height (also shown in figure 3.3). The scaling of the elements is done by varying the size of one block $b$, which is defined as the size of one method or field square. If $n_o$ is rounded down, then the amount of wasted space is due to extra height. There is no extra width, so $b$ was chosen such that $s_w \cdot n = e_w \cdot b$. Using this equality, and the equality that follows from the definition of $n_o$: $s_h / n_o = e_h \cdot b$, it is possible to calculate the wasted space as $s_w \cdot s_h \cdot e / (n_o + e)$. The calculation for the case where $n_o$ is rounded up is similar, and gives the following equation for the wasted space: $s_w \cdot s_h \cdot e / (n_o + e)$. Since in the second case $e > 0$,
we can say that in both cases \( w_s \leq s_w * s_h * e/n_o \). In other words, \( e/n_o \) is the maximum fraction of the screen space that is wasted due to the scaling problem.

![Figure 3.3: The layout of the rectangle (with fixed aspect ratio) containing all the elements on a single line. The top rectangle shows a layout where there is extra width, the bottom shows a layout where there is extra height.](image)

The rectangular blocks of size \( b \), create a regular structure. But if \( b \) is not an integer number, then it has to be rounded to the nearest integer for drawing, since drawing is done in pixels. If \( b \) is large enough, the difference due to rounding does not produce very profound visual artifacts, but if \( b \) is a small number, then the difference is relatively big, which creates very visible irregularities. If \( b \) is rounded down, an amount of wasted space \( w_b \) is created. If \( b \) is rounded up then the elements will not fit on the screen. Say \( b_o \) is the optimal blocksize, and \( b \) is equal to \( b_o \), rounded down to the nearest integer, then \( b_o - b \leq 1 \). The optimal screen usage, with respect to this problem, occurs when \( b_o \) is not rounded down. In this case the space on the screen occupied is \( e_w * e_h * b_o^2 \), if a size of \( b \) is used then the screen space occupied is \( e_w * b * b^2 \). The difference is the amount of wasted space, which is equal to \((b_o^2 - b^2) * e_w * e_h \).

Since \( b_o \leq b + 1 \), the amount of wasted space \( w_b \leq (2b + 1) * e_w * e_h \).

A third problem that arises with the layout described in the previous section is that packages would be cut where the border between the lines is. This would not be a big problem if it would be cut exactly between two classes. Enforcing this causes an amount of wasted space \( w_c \), which would mean suboptimal screen usage. An upper bound for the wasted space is \( w_c \leq s_h * e_w \) where \( e_w \) the width of the widest class and \( s_h \) is the height of the screen (see figure 3.4). It can be seen that this is an upper limit by looking at the optimal layout where the packages are split without regard to the location of the classes. If we now split the screen in two parts, where the rightmost part has width \( e_w \), then we can place all the classes on lines in the leftmost part. If a package should be split to fit in the leftmost part, it is now possible to place the class that would be split in two on that line and split the package after that class. The class will never fall off the screen since in the worst case the faultline in the class would fall exactly at the beginning of that class, and it has width that is smaller or equal to \( e_w \).

After some practical tests \( w = w_s + w_b + w_c \) was found small enough for practical purposes.
Alternatives to lower this bound result in unwanted visual aspects. To minimise $w_s$, the rectangularity of the blocks could be sacrificed. This leads to elements of varying aspect ratio, which would hamper the recognition of elements, and sets of elements. Another way to minimise $w_s$ would be to make the amount of methods and fields that fit on top of each other variable, such that the aspect ratio of $e_w : e_h$ changes to better fit the aspect ratio of $s_w * n : s_h/n$. Doing this also changes the appearance of classes and therefore that of packages, when the screen is resized, which would also make it harder for the user to associate a certain layout with a class or package. The third problem could be solved by using Linear Programming (LP) \cite{Vanderbei}, but this would significantly complicate the layout algorithm, and the amount of time needed for such a LP optimisation might seriously decrease responsiveness. The heuristic given here is relatively simple and response times of the current implementation are good. It is possible to distribute the wasted space across the entire visualisation, instead of letting it all accumulate at the bottom of the screen. This would draw less attention to the wasted space. This adaptation would not decrease the amount of wasted space, and would require a more complex drawing algorithm, which is why it has not been done.

\section{Class relation view layout}

Besides metrics, which are basically attributes of the source elements, it is useful to visualise relations between elements, described in section \ref{sec:metrics}. Showing these relations could be done by drawing arrows between elements in the layout given above. This would seriously clutter the display for most kinds of relations. To solve this problem a second level of zooming is added. On this level a class is visible in the centre of the screen with all its methods and fields, laid out as described in the previous section, but with room between the methods and fields such that they fill the allotted space. All classes that are related are also shown, the rest is hidden from view. A class is related to the centre class if the entire class, including methods and fields, contains any element related to an element of the centre class. These related classes are laid out to the left and the right of the centre class. If the relations connecting the classes are directed, then the classes that have a relation ending in the centre class are shown to the left, the rest is shown on the right side. Figure \ref{fig:related_classes} shows an example.
Chapter 3 Panoramix, a panoramic visualisation tool for metrics

of such a layout.

Figure 3.5: An example of the layout of a class and all related classes.

The edges of the relations of a class can also be drawn. For some relations the number of edges is such that drawing all edges will still result in a cluttered view. For these types of relations there is also the possibility to show only certain edges. One possibility is showing only the edges to and from selected elements. Another possibility is to show only external edges, which are edges starting or ending outside the centre class. Conversely it is possible to only show the internal edges of the centre class. Furthermore the external edges can be split up into edges coming from and edges going to the centre class (see figure 3.6).

Figure 3.6: The incoming, outgoing, and internal edges for a certain relation and class. When combined, the incoming and outgoing edges form the external edges. The external edges combined with the internal edges are all the edges of a relation.
Chapter 4

Visualised data

As mentioned in chapter 1, Panoramix will visualise metrics. Before showing how these metrics are visualised, it is good to see what they represent. Therefore section 4.1 will show some of the types of metrics that Panoramix can visualise. These metrics are all extracted using external programs. If other programs are found that can extract metrics for Java code, then these could also be visualised. The given list shows only metrics that can be imported at the writing of this text.

Another thing that Panoramix can visualise is relations between visualised elements. Relations are also extracted using external programs. A list of such relations can be found in section 4.2. As with the metrics, the given list shows only relations that can be imported at the writing of this text.

4.1 Metrics

A software metric is, as stated before, a quantitative measure of a certain aspect of (an element of) a software system. One of the first software metrics was the Cyclomatic Complexity Number (CCN) conceived by Thomas J. McCabe, on which he published a paper in 1976 [McCabe, 1976]. Since then many more software metrics have appeared, all of them useful, either in general or for measuring a specific aspect. The use of metrics can have different purposes, and the boundaries are not clear. Many metrics can be used for one purpose, and many purposes can employ the same metric. An example of metric usage is the estimation of bugs contained in certain code. To estimate the amount of time needed to adapt a program to new requirements is another example. Another use of metrics is to see which elements need to be optimised, or could benefit most from optimisation.

Section 4.1.1 handles basic metrics that measure code complexity, which can be used to asses adaptability, maintainability, and testability, for example. A special type of code complexity is handled in section 4.1.2, namely the informational complexity. This measure shows the complexity of code with respect to the information handled. There are also measures for the design structure of a program, which is handled in section 4.1.3. The metrics mentioned are extracted using Dependency Finder, JavaNCSS and RevJava.

4.1.1 Basic code complexity metrics

The simplest metric supported by Panoramix is the number of Non Commenting Source Statements (NCSS). The tool used to extract this number is JavaNCSS. The definition of a Source Statement used by this program is a sum of the number of package, import, class,

---

1http://depfind.sourceforge.net
2http://www.klee.de/clemens/java/javancss/
3http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm
interface, field, method, and constructor declarations plus the number of constructor invocations, labels, expressions, and code keywords (such as: if, while, break). Simpler, albeit less accurate, description of this number is given by the author on the JavaNCSS website. It says there that the NCSS “is approximately equivalent to counting ‘;’ and ‘{‘ characters in Java source files”. This number is very akin to the number of Lines Of Code (LOC). The number of LOC is calculated as the number of newlines in a piece of code. This number is influenced by the coding standard. If the convention is, for example, to put the starting { of an if, for and while statement on a new line, instead of on the same line, then the number of lines of code increases with the number of such statements present in the code. Comparing two pieces of code with a different coding standard would be biased.

Counting the number of statements gives a simple complexity metric. A somewhat more complex measure is the Cyclomatic Complexity Number (CCN) [McCabe, 1976]. The CCN measures the number of linearly independent execution paths through a piece of code, so each time that the execution path splits up the CCN counter is incremented by one. The theory of the CCN is well funded in graph theory. Given a graph of the control flow for a certain component, then the CCN is equal to \( E - N + p \), where \( E \), \( N \), and \( p \) are the number of edges, nodes, and attached components respectively. Most CCN extractors use another convention for the CCN, namely the number of decision points (if, for, while, etc.) plus one. Since the complexity of a piece of code is very closely related to the number of execution paths through that code, the CCN is a good measure for complexity. The fact that this metric originated in 1976, and is still in use today might be the best proof of the usability of this metric.

Another metric that has withstood the test of time is the Structural Fan-Out (SFO). The SFO is also sometimes called the span of control, which originates as a management term for the amount of people that report to one manager. This metrics was published in a book about structured design in 1979 [Yourdon and Constantine, 1979] together with the Structural Fan-In (SFI, see section 4.1.3). They where introduced as design heuristics. The SFO is derived from the static call graph, which is the graph showing which elements in a software system call other elements. The higher the SFO, the more methods are called. This makes the SFO a measure of the complexity of the code, since the behaviour of each method called has to be known in order to know what the behaviour of the calling method is.

A more complex metric is that of the Number Of Code Smells (NOCS). A code smell in code is a hint that there might be something wrong with the code. A list of types of code smells is given by Fowler et al. [Fowler et al., 1999]. An example is the duplication of code. If, for example, two methods are very similar, then they could be refactored into one method, resulting in better code since changes no longer have to be made to keep both behaving in the same way. Another example is a method that uses many features of another class, this method should probably be moved to that class. These Code Smells are hints that something might be wrong. A certain element that has a high NOCS (the terminology in the book is: “smells bad”), needs attention. The downside of this metric is that it is hard to calculate with a program, and that not all code smell types necessarily have the same severity.

## 4.1.2 Informational complexity metrics

A metric that gives insight into the amount of information that flows to and from a method is the number of parameters that a method has. In general, if a method has fewer parameters, less information flows to and from the rest of the system. There is, however, a fairly large bias on this interpretation of that number since not all parameters contain the same amount of information. There is another problem that this metric indicates: if a method takes a large number of parameters, then these parameters can usually be grouped into related sets, thus creating sets that represent objects that were not yet abstracted by the system. An example of this is a method that takes four integers to represent a rectangle, apparently the abstraction of a rectangle is missing in this system.

A method can call other methods, leading to the SFI and SFO metrics. A similar situation
exists when a method reads a field, which gives two metrics, namely the Number of Fields Accessed (NFA) for the accessing method, and the Number of Accessing Methods (NAM) for a field. These two metrics give more information about the amount of information that a method processes. The more fields a method accesses the more informationally complex a method is. As with the number of parameters, not all fields contain the same amount of information, which creates a bias in the interpretation of this number.

A standard for informational complexity by Henry and Kafura [Henry and Kafura, 1981] is defined as \((IFI \ast IFO)^2\), where IFI and IFO are the informational fan-in and informational fan-out respectively, which were defined as follows. The IFI is the amount of information in the input parameters plus the global data structures read. The IFO is the amount of information in the output parameters plus the global data structures updated. A difference should be made in reading and writing to a parameter or field. Given this definition of informational complexity, the number of fields accessed and the number of parameters do not give a complete picture. An approximation of the informational complexity can be calculated by combining the structural fan-in and structural fan-out with the number of parameters and the number of fields accessed.

### 4.1.3 Design metrics

A more controversial metric is the Inheritance Depth (ID). This is the number of inheritances a class is separated from the root class it is a descendant of. A general interpretation of this metric is, that the higher the ID, the more inherited methods there are to manage, which makes the class more complex. A low ID, on the other hand, could mean that the class does not use inheritance as a way to reuse methods, making the design less reusable. Chidamber and Kemerer [Chidamber and Kemerer, 1994] did testing of the ID metric and its use, and found that it was indeed a useful metric to use. Prechelt et al. [Prechelt et al., 2003] drew the opposite conclusion based on their own experiment, namely that “code maintenance effort is hardly correlated with inheritance depth, but rather depends on other factors”.

The SFI metric mentioned before is the amount of other elements that call a certain element. It can be said that the higher the SFI, the more reusable an element is. This makes the SFI metric well suited for evaluating the design. This metric and the SFO can be used as “indicators by which a structure may be examined for potential improvements” [Yourdon and Constantine, 1979]. The higher the SFO, the more complex a method is, as stated in section 4.1.1. This complexity is not only a part of the code complexity, but also of the design complexity. The more methods called, the more intertwined a method is with other elements, which has a negative impact on adaptability of the design.

Several of the metrics mentioned before are derived from a certain dependency of one element upon another. For instance the ID is derived from the inheritance of one class from the other. Without the parent class, the inherited class cannot function. The Number of Incoming Dependencies (NID) and the Number of Outgoing Dependencies (NOD) metrics are measures of such dependency relations. The more dependant a certain element is on other elements, the less reusable it is. And the more elements there are that depend on a certain element, the harder it will be to change that element. To track these dependencies automatically, the program Dependency Finder has been created. This program extracts several such dependencies, such that given these dependencies other dependencies can be inferred. For instance if a method depends on another method, then the class containing that method also depends on that method, and on the class that holds that method.

### 4.2 Relations

Relations are useful to show a structure in the system visualised, other than the basic structure of package, class, and method which is the basis of the visualisation. These different structures can also interact with the metrics. It might be of importance how the value of
a certain metric for an element relates to the value of that metric for related elements. In other words, two elements that are related can influence each other in some way, so it could also be important to see how the metrics relate. In general, if two elements are related, then there is a dependency of some sort between the two. The relations mentioned here are all extracted using the tool Dependency Finder. When applying these relations to a system, it is possible that there is a relation between two elements where one of these elements is not inside the visualised system. Such a relation between two elements is discarded since it reflects no relation edge inside the visualised system.

4.2.1 Call relation

The first relation for which the derived metrics are mentioned in the previous section is the call relation. A method can call another method, which creates a call relation. The reasons why the SFO and SFI are important also apply to this relation. The call relation is a good indication of the control flow within the system. For example, the closure over the call relation for a certain element shows all the elements that the control can reach from that element. That is not to say that the control will actually reach all those elements. The call graph mentioned here is a static call graph, as opposed to a dynamic call graph. A dynamic call graph is constructed from the calls that were done during execution of a system, and not extracted from the source as the static call graph is.

4.2.2 Inheritance relation

Another relation mentioned already in the previous section is the inheritance relation. If a class inherits from another class, then it inherits all its non-private methods. So the metrics pertaining to these methods are of influence to the inheriting class. The closure over the inheritance relation contains all the classes that this class inherits from, which all can contain methods that the first class inherits.

4.2.3 Access relation

Similar to the call relation, where a method calls another method, is the access relation, where a method accesses a field. If a method accesses a field of another class, then this method’s functioning is closely tied to the internal workings of the other class. A method is also related to all other methods that access a field it also accesses.

4.2.4 Use relation

A class can be used inside a method, when the method handles an instantiation of that class. This happens by definition if the method calls a method of that class, or accesses a field. There are also cases where a class is used without methods being called or fields being accessed. For example, if an object of that class type is extracted from one list and added to another. If a certain method uses a class, then the metrics of that class can be of importance, since the method relies on that class.

\footnote{http://depfind.sourceforge.net}
Chapter 5

Metric visualisation

After the metrics are computed, statistical analysis can be performed, for instance calculating the mean, outliers, and standard deviation. Another approach, the one used for Panoramix, is to visualise these metrics. The structure used for this visualisation was already explained in chapter 3. This chapter further explores the visualisation of the metrics on the given structure. The metrics for all the elements in the visualisation fall in a value range. What is needed is a mapping form this value range to certain visual aspects. The first of these visual aspects handled in section 5.1 is colour. Section 5.2 handles other visual aspects that might be used to represent values, which can be combined with colour. After that section 5.3 handles some ways of preprocessing the values before displaying them. Finally section 5.4 concludes with what was chosen to be implemented.

5.1 Colour coding

Given the structure in chapter 3 there are several ways to display a single value. The one most used in diagrams is colour coding. This means that each value has a corresponding colour. It is preferable that the colours vary continuously over the range that is mapped to a certain value range, in this way a difference between two colours represents the same difference in value throughout the range. This can be achieved in several ways. A person can choose a series of colours such that they form a visually continuous range. Colours can also be automatically picked from a certain colour space, such that the distance between the colours is equal. A colour space is a multidimensional (mostly three-dimensional) space in which all colours are placed. Dependant on the number of dimensions of a colour space there are several axes, each representing a fundamental element of a colour. An example is the RGB colour space used by most (computer) monitors. This colour space is based upon the fact that all colours can be split into three components: red, green, and blue, which when added additively form a colour that the human eye cannot distinguish from the original colour. These three components are the main axes of the colour space. It is then, for example, possible to obtain a range of colours by varying the red component and keeping the rest of the colours at their minimum value, which creates a colour range that varies between red and black.

The RGB colour space is not the best colour space to use for continuous colour ranges. The RGB colour space is not perceptually uniform, meaning that a given change in value does not correspond to the same perceptual difference. An alternative to this colour space is the HSB colour space, which is more uniform. The axes of the HSB model are Hue, Saturation and Brightness. It is now fairly easy to get a list of colours that vary (almost) continuously, for example by picking colours for different values of for instance the hue, while keeping the other two values constant. This can also be done for example for the saturation and the brightness.
Chapter 5  Panoramix, a panoramic visualisation tool for metrics

Figure 5.1: A slice from the CIE L*u*v* colour space where L = 50, u varies in [-40..160], and v varies in [-120..60]. It is converted to the RGB colour space using a CIE observer and Illuminant of 2° and D65 (daylight) respectively. The area outside the black lines are approximations of the CIE L*u*v* colours which cannot be displayed as RGB colours.

Figure 5.2: A colour list derived from the slice of the CIE L*u*v* colour space in figure 5.1

A drawback of the HSB colour space is that it is not completely perceptually uniform. There is a colour space, created by the Commision Internationale de l’Eclairage (CIE, International Commission on Illumination) to represent all visible colours, that is perceptually uniform. This colour space is the CIE L*u*v* (CIELUV) colour space (see figure 5.1). Its primary axes are L, u, and v, where L stands for Luminance, the u and v parameters do not represent an intuitive value. As an example, take the slice of the colour space given in figure 5.1, it is possible to pick colours along the border of the image from the bottom left to the top right, going through the top left. If the colours are selected from equidistant points then perceptual difference between the colours is equal due to perceptual uniformity (see figure 5.2).

It would be possible to encode two values in one colour. A problem that arises when encoding two-dimensional data into a single colour is the amount of information that can be transferred. The bottleneck here is easily locating an element of a certain colour. It is preferable for the user to be able to use preattentive processing [Healey et al., 1996] when looking at the resulting visualisation. Preattentive processing means that certain aspects (in this case colour) are processed rapidly and accurately by the human visual system. If elements of a certain colour are preattentively processable this means that the time it takes to locate those elements is not strongly dependent on the total number of elements. In other words, the user does not have to search all the elements to see if they are of the target colour, the target element stands out visually. There is research indicating that the number of different colours that are preattentive is approximately seven [Healey, 1996, Miller, 1956]. So if we have a two-dimensional colour signal that we want to be preattentively processable it is possible for a user to distinguish approximately three ($\sqrt{7} \approx 2.6$) different values for each dimension, which is not enough for our purpose. Before Healy and his colleagues published their theory on preattentive processing [Healey et al., 1996], Wainer and Francolini tested human understanding of Two-Variable Colour Maps [Wainer and Francolini, 1980]. Their findings corroborate that the performance of two-variable colour maps is, as they say, “a lot like that expected of a dog walking on its hind legs”. The low level processing of colour is not good enough for our purposes, but if we are less stringent and drop this wish then this visualisation could still be useful. As Wainer and Francolini say: “Thus, the only purpose for [a two-variable colour map] would be for its virtues at the intermediate or superior
level”. For this purpose research has been done on colour maps for multidimensional data \cite{Rheingans, 1992, Pham, 1990, Robertson and O’Callaghan, 1986}.

5.2 Multiple display dimensions

Displaying a single metric gives an overview of that metric. But there might be a correlation between different metrics, which is not visible to the user if only one metric at a time is visualised. This section handles the display of \(N\) different types of values in one view using \(N\) display dimensions. We will explore the possibilities of a visualisation method that use two dimensions simultaneously (\(N = 2\), one being colour. It might be possible to combine some of these methods to provide visualisations for \(N > 2\), determining if this is possible has not been explored in this project. Before a list of possible techniques is given it might be useful to investigate what we want from our visualisation of multiple values. It is desirable for a multivariate visualisation method to make it possible for the user to correctly estimate the separate values that are concurrently displayed (local processing, \cite{Wickens, 1992}), and the user also sees an emergent feature \cite{Pomerantz et al., 1977, Kubovy and Pomerantz, 1981} in the visualisation. An emergent feature is a global property that can be seen when looking at the whole, but is not evident when looking at each part in isolation.

Related to the use of colour is the use of textures. Textures are commonly used to transfer information, for example as an alternative to colour visualisations for media that do not support colour. Texture is used in flow visualisation for texture based flow visualisation \cite{Max and Becker, 1995}. The application of textures in representing multivariate datasets has also been researched \cite{Healey and Enns, 1999, Healey et al., 2004, Holten, 2005}. It is safe to say that textures are useful, but there are also limitations. The biggest limitation with regard to this project is that it requires a relatively large amount of space, compared to colour. A texture can be recognised by a repetition of a certain pattern over an area. If the area is too small then there is no room for repetition, which makes textures useless.

Up to now we have only considered 2D visualisations. A logical step would be to expand the visualisation into a 3D one. 3D visualisations are well established. They have been used and researched for a fairly long time. An inherent problem to visualising data in 3D is occlusion, this occurs when one object obstructs the view of another. This means that it is almost impossible to get an overview of the visualisation without moving the point of view. This is directly in conflict with the idea of a static, panoramic visualisation, which is the goal of this project. This is not to say that the concept of a 3D visualisations is not useful, we believe that it is just not useful in this context. Another, less serious, problem of a 3D visualisation is the influence 3D visualising has on the display of colour and textures. Since a 3D display uses changes in colour/texture to create the illusion of depth, colour and texture change with different lighting. This problem can generally be avoided by a good choice in colours, textures, and lighting however.

It is also possible to use the physical ‘fourth dimension’, otherwise known as time, as extra dimension. In signal processing, time is often used for combining signals, this process is known as Time Division Multiplexing (TDM) \cite{Stallings, 2004}. To use this method, the static visualisation should be changed in an animation of some sort. This dimension is best used when visualising a value that, in reality, also changes through time. In program visualisation the use of animation is mostly used to show the runtime behaviour of a program or the evolution of the code through time, results vary though. Tudoreanu \cite{Tudoreanu, 2003} said about the use of animation to show algorithm execution that there were “mixed results of empirical studies”. As mentioned before, the main idea was to create a static visualisation, which directly opposes the use of time as an extra dimension. And even if we would overlook this objection, it is still questionable how good the visualisation would be to display some value as changing through time that, in reality, does not. In other worlds, it is questionable if this technique is applicable to a wide range of metrics.

Besides the colour and the position of an element, it is also possible to vary its form.
In this way it is possible to create a very rudimentary icon that can convey information. There are certain limits in the ways in which the shape can be varied. Since the colour of a shape also conveys information, we do not want to change the perception of that colour by changing the shape. If the total area of a shape changes too much, then the total amount of colour visible will also change, which will distort the colour. To implement such a shape in 3D, Shaw [Shaw et al., 1999] used a three dimensional superellipse [Weisstein, 1999]. The superellipse is not suited for small two dimensional shapes because there is no clear linear ordering in distinct two dimensional superellipses. There are other forms that do have a clear linear ordering, a regular $N$-sided polygon for example. Another possibility is to use the orientation of a shape to convey information. In this case the form itself would stay the same, but it would be rotated.

5.3 Preprocessing

Up to now, only the mapping of the data to visual dimensions was mentioned. The visualisation of the data can be preceded by a step where the values that are to be displayed can be changed in some way. One of the possibilities is to change the range of values to display. For example if the user is only interested in the outliers, then it would be possible to change the range such that all values below a certain value have the same value. Conversely, if the user is not interested in the outliers then the upper bound of the range can be lowered.

The methods mentioned in the previous sections are all oriented towards displaying patterns to the user. Another possible preprocessing step is to combine several values into one value through mathematical functions. If a user is able to discern a pattern in a display showing multiple values, the next step would be to quantify this pattern in the form of a mathematical function. This function can then produce a single value for all the elements in the visualisation, and this value can then be visualised. As with all the methods mentioned, there are also problems with this one. The problem is that, as already mentioned, the user first has to have an idea of a pattern before we can use this method. This method is widely used to create metrics, since most of the metrics are already based on some mathematical function which contains primitive metrics. This type of preprocessing tool can be used in the confirmatory phase [Kotz et al., 1982] of Exploratory Data Analysis (see section 1).

5.4 Conclusion

Colour coding (see section 5.1) is best suited for displaying a single metric because it is preattentively processable, and because it has proven itself in almost all data visualisations. For the display of a second metric (see section 5.2) at the same time, shape is best suited for the purpose of this visualisation. Of the other options, using animation and using a 3D visualisation might also deliver results, but these do not fit with our goal of a static visualisation. To manipulate the data before visualising (see section 5.3) both adapting of the range and using mathematical functions to create new metrics are suited for our purposes.

The options listed in this chapter are not meant to be conclusive, it is a list of all the possibilities that have been explored for Panoramix. Some of the methods mentioned are already used by other programs (see section 2), and others could be used in yet to be created programs. Panoramix currently uses colour to visualise one metric. Optional is the use of shapes to visualise a second metric, more specifically directed shapes, and N-gons. Both preprocessing methods mentioned in section 5.3 are also implemented.
Chapter 6

Application of Panoramix

This chapter will give an example of an application of Panoramix. The code base used for the first example is the source code for the `java.util` package (and its subpackages) from Java version 1.4.2. This code was extracted from the Java 2 SDK 1.4.2, SCSL source. Table 6.1 shows some statistics of this code base. Unless noted otherwise, all the visualisations that can be seen in this chapter where created and exported to Scalable Vector Graphics (SVG) format by Panoramix, the visualisation on the users screen might differ slightly from this SVG version.

<table>
<thead>
<tr>
<th>Packages</th>
<th>Classes</th>
<th>Functions</th>
<th>NCSS</th>
<th>Javadocs</th>
<th>per</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>123</td>
<td>1843</td>
<td>17580</td>
<td>1768</td>
<td>Project</td>
</tr>
<tr>
<td>20.50</td>
<td>307.17</td>
<td>2930</td>
<td>294.67</td>
<td>294.67</td>
<td>Package</td>
</tr>
<tr>
<td>14.98</td>
<td>142.93</td>
<td>14.37</td>
<td>9.54</td>
<td>0.96</td>
<td>Class</td>
</tr>
</tbody>
</table>

Table 6.1: Some quantitative information on the code base, obtained with JavaNCSS. NCSS are Non Commenting Source Statements. Javadocs are the number of Javadoc comment blocks.

6.1 Complexity measure

The basic visualisation of NCSS (see section 4.1.1) over the code base can be seen in figure 6.1. Note that a package, a class and a field element do not have any amount of NCSS since that number is not defined for those elements. These elements are therefore coloured dark grey. The class that is highlighted with a white border has some very brightly coloured methods, which are also highlighted. The class is the `Pattern` class from the package `java.util.regex`. The highlighted methods, from top to bottom, are `escape` (see appendix A.1) and `group0` (see appendix A.2), the first is responsible for parsing an escape sequence from a regular expression, the second parses a regular expression group. They attract attention, which they should, since the number of statements for those methods is exceptionally large, compared to most of the other methods in the visualisation.

Next the Cyclomatic Complexity Number (CCN) will be handled. The CCN is, put simply, the number of decision points in the code as explained in section 4.1.1. The higher the NCSS, the higher the CCN is likely to be, since more statements would probably mean more statements that form decision points. If this is true, then CCN is not a very enlightening metric in our visualisation. To check this assumed link between NCSS and CCN, they are

1http://www.sun.com/software/communitysource/j2se/java2/download.xml
2http://www.kclee.de/clemens/java/javancss/
both shown in one visualisation in figure 6.2. The visualisation shows the expected relation, the more red an element is, the more sides the N-gon of the element has. The highlighted methods also clearly show this relation. Another thing that catches the eye for the selected elements is that although there seems to be a proximity in the colouring of the two elements, there is a bigger difference in the shapes (one is a pentagon and the other an octagon). Checking the actual amounts confirms what is seen, the escape method has a CCN of 117 and 164 NCSS, while group0 has values of 48 and 132 respectively.

This dependency of CCN on NCSS means that visualising the CCN would not give us what we want to see. It would seem that we are more interested in the CCN per NCSS. Gill and Kemerer [Gill and Kemerer, 1991] came to a similar conclusion in 1991, when they tested a new metric based on the CCN. This new metric, named the Complexity Density Ratio (CDR, also referred to as Cyclomatic Density, or Decision Density), was defined by them as the CCN per thousand lines of executable code. Panoramix also offers a way of interactively creating a new metric based on existing ones. Here the new metric, which will be named CDR after the Complexity Density Ratio, will be computed as the percentage of decisions per source statement (100 \times (CCN − 1)/NCSS). The CCN is decreased by one since we want the number of decisions, not the number of execution paths through the code. Gill and Kemerer tested this new metric on several “small customised programs primarily for real-time defence applications”. They concluded that the CDR was a good indication for the time needed to do maintenance on one of these programs. So their conclusion was that when the CDR is higher, the complexity of a program is higher.

The CDR can be seen visualised in figure 6.3. It can be seen here that the method escape
does indeed have a higher CDR than \texttt{group0} (respectively 71 and 36). We can also see that
the CDR is more regularly distributed than CCN and NCSS, there are no clear outliers.
It is possible to see that some classes have many complex methods, and others have very
few. Another thing that is evident is the very regular distribution of complex methods in a
class in the second row (the 17th from the left). This is the \texttt{Arrays} class from the package
\texttt{java.util}, which contains several methods for manipulating arrays. The series of methods
that all have the same colour are all overloaded versions of the method \texttt{equals}. A look at
the code of these reveals that all of them consist of a copy of the same code, with the only
difference that one compares integers, while another other compares floats for example. Most
of the methods of that class are overloaded methods providing the same functionality (for
example swapping) for different basic types.

The Structural Fan-Out (SFO) is also a complexity metric (see section \ref{sec:complexity-metrics}), just as
CDR is. In figure \ref{fig:sfo-arrays} the SFO is visualised for \texttt{java.util}. Comparing this picture to \ref{fig:cdn-arrays}
shows that the complexity measures SFO and CDR are both high for the selected methods.
This encourages the idea that these methods are indeed complex. Furthermore the selected
class (named \texttt{Pattern}) seems to have a relatively large amount of methods with a high SFO,
and also a large CDR. This encourages the thought that entire class is more complex, not
just the selected methods.

Figure 6.2: The same as figure \ref{fig:cdn-arrays} but now with regular N-gons indicating the CCN. The
minimal number of sides of the N-gon is 3, indicating a CCN of 0. The maximal number of
sides is 8 representing a CCN of 117.
Chapter 6 Panoramix, a panoramic visualisation tool for metrics

Figure 6.3: The visualisation of java.util, with the amount of red indicating the CDR. The minimum value is 0 and the maximum value is 86.

Figure 6.4: The visualisation of java.util, with the amount of red indicating the SFO. The minimum value is 0 and the maximum value is 28.

6.2 Relations

The previous section showed that the SFO metric can be used as a complexity measure. As mentioned in chapter 4, the SFO metric is derived from the call relation. This call relation can also be visualised by Panoramix, such a visualisation is shown in figure 6.5. This visualisation shows all the classes that have a call relation with the Pattern class. One of the first things that can be seen here is that the left side of the screen is empty. This means that there are no incoming method calls into the Pattern class. An explanation for
this is that the Pattern class is a public class in the java.util.regex package, which is meant to be used as a regular expression library. This means that no calls are made to this class from inside the project. The group0 method, which is highlighted, calls many small classes. Closer inspection reveals that all of these classes are inner classes of the Pattern class. All the calls are calls to the constructor of these inner classes, which is the topmost method in all these classes. The one exception is the Node inner class. This class is a node in a tree representing a regular expression.

![Figure 6.5: The visualisation of the class java.util.regex.Pattern, showing the same metric as in figure 6.4 and additionally showing all the call relations for the highlighted method group0 as arrows.](image)

Looking at the access relation (see figure 6.6), it can be seen that there is a cyclic access relation, on a class level, between Pattern and its Node inner class. A class reading a field of another class is in general considered bad practice, since this leads to a dependency on the internals of another class. This might be mediated by the fact that the Node class is an inner class of the accessing class. The dependency of Pattern on the internals of Node combined with the dependency of Node on the internals of Pattern suggests a very close relation between the two. Another class closely related to Pattern is the Matcher class. Figure 6.5 shows that Pattern calls methods of Matcher, and figure 6.6 shows that Matcher has methods that read fields of Pattern. So looking at the class Pattern should be accompanied by a closer look at both the classes Node and Matcher.

### 6.3 Patterns

This section will show an example of patterns in the source code for jEdit\(^3\) version 4.2. jEdit is an open source text editor developed in Java. The source code contains more than simply the jEdit source, it also contains sources for BeanShell scripting\(^4\), the ASM java bytecode manipulation library\(^5\), and a custom installer, among other things.

In figure 6.7 a visualisation of all the packages contained in the jEdit source can be seen. The packages that can be seen have different patterns with respect to the Number Of Code

---

\(^3\)http://www.jedit.org  
\(^4\)http://www.beanshell.org/  
\(^5\)http://asm.objectweb.org/
Chapter 6 Panoramix, a panoramic visualisation tool for metrics

Figure 6.6: The visualisation of the class `java.util.regex.Pattern`, showing with red the number of accessors (minimum amount is 0, maximum amount is 39). The relation shown is the access relation (see section 4.2), for which only external edges are shown. The highlighted classes are both the `Node` inner class of the package `Pattern`.

Smells (NOCS, see section 4.1.1). The first package that is selected is the `bsh` package, the main package in the BeanShell code base. This package contains two classes that stand out because of their size and intense colouring, namely the first and last classes that have a width of more than three methods and fields. These are the `Parser` and `ParserTokenManager` classes. The names both hint at a relation with parsing. When looking at the source of these classes it can be seen that both these classes are generated using a parser-generator. It seems that this generated code has a relatively large number of code smells, which can be explained by the fact that this code was not made by a human programmer.

The package `org.gjt.sp.jedit` shows a different pattern. The members (methods and fields) of classes from that package do not have a notably different pattern from most other members in the visualisation. But the class elements in that package do have a relatively high NOCS, compared to other classes. This could be because the type of code smells used for classes are different from those used for members. A code smell for a class is derived on a class level, that is in relation with other classes, while code smells for members are on a member level. The package `org.gjt.sp.jedit` is a very central package, containing for example the `Buffer` class (the one with the most members) which represents a document being edited, and the `jEdit` class (the one with the second most members) which seems to be the main class in the application. The central position of this package probably means that they have a more complex interaction with other classes, therefore explaining the large amount of NOCS for these classes, but not for there members. To check this assumed position of the classes in this package, we can look at the number of elements depending on these classes (NID, see section 4.1.3). Figure 6.8 shows that at least two classes from the `org.gjt.sp.jedit` package have relatively very high NID. The class that has a very low NID for the `jEdit` class, but the assumption is that this class is still a key class in the hierarchy. To check this we look at the Structural Fan-In metric (SFI, see section 4.1.3). We can see in figure 6.9 that a relatively high amount of methods of the `jEdit` class have a high SFI. It even contains the method `getProperty` which has a SFI of 236, which is the second highest amount of SFI, next only to the static `log` method from the `Log` class.

The third selected package in figure 6.7 shows another kind of pattern. A lot of the classes
Figure 6.7: The visualisation of the jEdit, showing with red the NOCS (minimum amount is 0, maximum amount is 8). The highlighted packages are bsh, org.gjt.sp.jedit, and org.gjt.sp.jedit.options.

in that package have a similar pattern of NOCS values for their methods. All the classes for which the second and third method (in reading order) have a high value for NOCS are inherited from the AbstractOptionPane which resides in the org.gjt.sp.jedit package. This pattern can be seen more clearly when looking at all the classes inherited from this AbstractOptionPane class, as can be seen in figure 6.10. These methods all override the same abstract methods of their parent class, namely the _init and _save methods. The _init method “should create and arrange the components of the option pane and initialise the option data displayed to the user”, as described in the Javadoc comment of this method. The _save method “should save any properties being edited in this option pane”. So apparently, the implementations of these methods all lead to a similar amount of code smells. This indicates that the functionality could be hard to implement without violating certain code smell standards.
Figure 6.8: The visualisation of the jEdit, showing with red the NID (minimum amount is 0, maximum amount is 433). The highlighted classes are View, Buffer, and jEdit, from the package org.gjt.sp.jedit.
Panoramix, a panoramic visualisation tool for metrics

Chapter 6

Figure 6.9: The visualisation of the jEdit, showing with red the SFI (minimum amount is 0, maximum amount is 237, all values above 85 have the same red value). The highlighted method is `getProperty` of the class `jEdit` which has SFI value 236.

Figure 6.10: The visualisation of inheritance relation for the `AbstractOptionPane` class. The metric visualised is the same as in figure 6.7.
Chapter 7

Final remarks

7.1 Conclusion

The aim of this project was to create a panoramic visualisation tool for metrics. Specifically created so that it could display metrics for medium to large projects. This visualisation would then assist the user in locating complex elements in the code. The use case in chapter 6 shows that Panoramix can visualise such a system, and that it is also possible to draw conclusions from this visualisation. The visualisation indeed assists in finding complex elements.

The use case also shows that the visualisation can show patterns in the metrics. These patterns not only show the complexity of certain elements, but more generally the complexity of the design, or at least the implementation of that design. The aim was to give a view on the source code that would not impose a structure on the user that is too high level. Even though a high level design view was not created, the design can be deduced by the user, based on the visualisation.

There is also the possibility of formalising the patterns found into formulae, which form new metrics. These newly created metrics can then be compared to existing, and proven, metrics, or to the experience of the user with a certain code base. These comparisons can then reveal if this new metric provides extra information. This is the reverse of the original use of Panoramix, where the code base was the object to be tested and the metrics where clearly defined.

The abstraction between metrics and the code base provides for a program that is easily expandable with other metrics from external sources. Since the visualisation structure is extracted from the source code, and is used for the visualisation of all the different metrics, all metrics can be combined independent of their source.

The examples shown in chapter 6 are from programs that where chosen without prior knowledge of if these examples would yield results. They where chosen relatively at random, which suggests that the program is applicable to a wide variety of programs. It has been previously used with other code bases, in some cases upon request. All these experiences have led us to believe that the concept of metric visualisation, and specifically metric visualisation as done with Panoramix, could be very helpfull in code maintenance, and development.

7.2 Further work

Chapter 6 shows an example application of Panoramix. This use case shows that it is possible to infer things about the source code. A next step would be to measure the effect of Panoramix objectively. Therefore several tests on a real world code base with several users should be devised, to see if the amount of effort needed to locate key elements is less when using Panoramix. To do this, there would have to be a goal for the users to reach which is clearly measurable. For example the task of finding as much bugs as possible. Such a task
has to be objectively rated, for this to be possible there has to be a clear definition of a bug. Furthermore there has to be an alternative way of inspecting the code, preferably with an existing metric based tool that does not visualise the source, to see what the benefits are of visualising metrics.

The metrics are obtained from external sources, and a data visualisation is always very dependent on the quality of the data visualised. The metrics chosen have, as can be read in section 4.1 a good theoretical background. There are many metrics besides the ones mentioned in this text, some of them might be better interpretable when visualised. These metrics could also be used in Panoramix.

Given a good test for the effectiveness of Panoramix, as mentioned above, it is also possible to test other types of visualisations for a specific task. The assumption that a static panoramic view is beneficial to the understanding of a system can be checked. This assumption is based on the model of the working memory by Baddeley [Baddeley, 1986] (see chapter 1). Other ways of displaying multiple metrics, as described in chapter 5 could be tested. The advantage of having a three dimensional visualisation showing two metrics, for example, might outweigh the extra cognitive effort needed to interpret such a visualisation for some tasks.
Appendix A

Example source code

The source code in this appendix is a part of the code used as an example in chapter 5 which is a part of the source code of the \texttt{java.util.regex.Pattern} class, extracted from the Java 2 SDK 1.4.2. SCSL source. Appendix A.1 gives the source code for the method \texttt{escape}, and appendix A.2 gives the code for the method \texttt{group0}.

A.1 escape

```java
/**
 * Parses an escape sequence to determine the actual value that needs
 * to be matched.
 * If -1 is returned and create was true a new object was added to the tree
 * to handle the escape sequence.
 * If the returned value is greater than zero, it is the value that
 * matches the escape sequence.
 */
private int escape(boolean inclass, boolean create) {
    int ch = skip();
    switch (ch) {
        case '0':
            return o();
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            if (inclass) break;
            if (groupCount < (ch - '0'))
                error("No such group yet exists at this point in the pattern");
            if (create) {
                root = ref((ch - '0'));
            }
            return -1;
        case 'A':
```
if (inclass) break;
if (create) root = new Begin();
return −1;
case 'B':
if (inclass) break;
if (create) root = new Bound(Bound.NONE);
return −1;
case 'C':
break;
case 'D':
if (create) root = new NotCtype(ASCII.DIGIT);
return −1;
case 'E':
case 'F':
break;
case 'G':
if (inclass) break;
if (create) root = new LastMatch();
return −1;
case 'H':
case 'I':
case 'J':
case 'K':
case 'L':
case 'M':
case 'N':
case 'O':
case 'P':
break;
case 'Q':
if (create) {
    // Disable metacharacters. We will return a slice
    // up to the next \E
    int i = cursor;
    int c;
    while ((c = readEscaped()) != 0) {
        if (c == '\\') {
            c = readEscaped();
            if (c == '\E' || c == 0)
                break;
        }
    }
    int j = cursor−1;
    if (c == '\E')
        j==;
    else
        unread();
    for (int x = i; x<j; x++)
        append(temp[x], x−i);
    root = newSlice(buffer, j−1);
}
return −1;
case 'R':
break;
Appendix A
Panoramix, a panoramic visualisation tool for metrics

case 'S':
    if (create) root = new NotCtype(ASCII.SPACE);
    return -1;
case 'T':
case 'U':
case 'V':
    break;
case 'W':
    if (create) root = new NotCtype(ASCII.WORD);
    return -1;
case 'X':
case 'Y':
    break;
case 'Z':
    if (inclass) break;
    if (create) {
      if (has(UNIX_LINES))
        root = new UnixDollar(false);
      else
        root = new Dollar(false);
    }
    return -1;
case 'a':
    return '\007';
case 'b':
    if (inclass) break;
    if (create) {
      if (has(UNIX_LINES))
        root = new Bound(Bound.BOTH);
      return -1;
    } else
    return \033;
case 'c':
    return c();
case 'd':
    if (create) root = new Ctype(ASCII.DIGIT);
    return -1;
case 'e':
    return '\033';
case 'f':
    return '\127';
case 'g':
case 'h':
case 'i':
case 'j':
case 'k':
case 'l':
case 'm':
    break;
case 'n':
    return '\n';
case 'o':
case 'p':
case 'q':
    break;
case 'r':
    return '\r';
case 's':
case 't':
case 'u':
case 'v':
    return '\v';
case 'w':
case 'x':
    return '\x';
case 'y':
case 'z':
A.2 group0

/*
 * Parses a group and returns the head node of a set of nodes that process
 * the group. Sometimes a double return system is used where the tail is
 * returned in root.
 */
private Node group0() {
    Node head = null;
    Node tail = null;
    int save = flags;
    root = null;
    int ch = next();
    if (ch == '?') {
        ch = skip();
        switch (ch) {
            case ':': // (?:xxx) pure group
                head = createGroup(true);
                tail = root;
                head.next = expr(tail);
                break;
            case '=': // (?!xxx) lookahead
                head = createGroup(true);
                tail = root;
                head.next = expr(tail);
                break;
            case '!': // (?=xxx) and (!?xxx) lookahead
                head = createGroup(true);
                tail = root;
                head.next = expr(tail);
                break;
        }
    }
    return -2;
}
Appendix A  Panoramix, a panoramic visualisation tool for metrics

```java
head = tail = new Pos(head);
} else {
    head = tail = new Neg(head);
}
break;
case '>'; // (?>xxx) independent group
    head = createGroup(true);
    tail = root;
    head.next = expr(tail);
    head = tail = new Ques(head, INDEPENDENT);
    break;
case '<'; // (?<xxx) look behind
    ch = read();
    head = createGroup(true);
    tail = root;
    head.next = expr(tail);
    TreeInfo info = new TreeInfo();
    head.study(info);
    info.maxValid == false {  
        return error("Look-behind group does not have "+" an obvious maximum length");
    }
    if (ch == '=' ) {
        head = tail = new Behind(head, info.maxLength, info.minLength);
    } else if (ch == '!' ) {
        head = tail = new NotBehind(head, info.maxLength, info.minLength);
    } else {
        error("Unknown look-behind group");
    }
    break;
case '1'; case '2'; case '3'; case '4'; case '5'; case '6'; case '7'; case '8'; case '9';
    if (groupNodes[ch-'0'] != null) {
        head = tail = new GroupRef(groupNodes[ch-'0']);
        break;
    }
    return error("Unknown group reference");
case '$'; case '@';
    return error("Unknown group type");
default: // (?xxx) inlined match flags
    unread();
    addFlag();
    ch = read();
    if (ch == ') ') {
        return null; // Inline modifier only
    }
    if (ch != ':') {
        return error("Unknown inline modifier");
    }
    head = createGroup(true);
    tail = root;
```
head.next = expr(tail);
break;
}
} else { // (xxx) a regular group
    head = createGroup(false);
    tail = root;
    head.next = expr(tail);
}
accept(')', "Unclosed group");
flags = save;
// Check for quantifiers
Node node = closure(head);
if (node == head) { // No closure
    root = tail;
    return node; // Dual return
}
if (head == tail) { // Zero length assertion
    root = node;
    return node; // Dual return
}
if (node instanceof Ques) {
    Ques ques = (Ques) node;
    if (ques.type == POSSESSIVE) {
        root = node;
        return node;
    }
    // Dummy node to connect branch
    tail.next = new Dummy();
    tail = tail.next;
    if (ques.type == GREEDY) {
        head = new Branch(head, tail);
    } else { // Reluctant quantifier
        head = new Branch(tail, head);
    }
    root = tail;
    return head;
} else if (node instanceof Curly) {
    Curly curly = (Curly) node;
    if (curly.type == POSSESSIVE) {
        root = node;
        return node;
    }
    // Discover if the group is deterministic
    TreeInfo info = new TreeInfo();
    if (head.study(info)) { // Deterministic
        GroupTail temp = (GroupTail) tail;
        head = root = new GroupCurly(head.next, curly.cmin, curly.cmax, curly.type,
                             ((GroupTail)tail).localIndex,
                             ((GroupTail)tail).groupIndex);
        return head;
    } else { // Non-deterministic
        int temp = ((GroupHead) head).localIndex;
        ...
Appendix A

Panoramix, a panoramic visualisation tool for metrics

Loop loop;
if (curly.type == GREEDY)
    loop = new Loop(this.localCount, temp);
else // Reluctant Curly
    loop = new LazyLoop(this.localCount, temp);
Prolog prolog = new Prolog(loop);
this.localCount += 1;
loop.cmin = curly.cmin;
loop.cmax = curly.cmax;
loop.body = head;
tail.next = loop;
root = loop;
return prolog; // Dual return
}
else if (node instanceof First) {
    root = node;
    return node;
}
return error("Internal logic error");
Bibliography


Bibliography

Panoramix, a panoramic visualisation tool for metrics


