TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

Master's Thesis

Visualization of File Relations in Software Systems with Adjacency Matrices

By
Rob Goud

Supervisors:

Prof. dr. ir. Jarke J. van Wijk (TU/e)
Drs. Paul Jansen (TIOBE Software)

Eindhoven, April 2006

Chapter 1

Abstract

Over time there may have arisen unwanted differences between the contrived design and the actual implementation of a software system. To detect the differences, it is necessary to be aware of the system's current structure and behavior. Companies usually don't have the current information at hand, because software code ages, systems grow larger and employees (knowledge) leave. It is quite tedious to manually reconstruct a software system's structure and behavior by reasoning about the system and by unraveling the system's code. That is why aids are needed to get a better and easier insight in a system's structure and behavior.

Graph drawings are one way to visualize system information, i.e., the interdependencies between components. The most common graph representation is the node-link diagram. However, the graphs can grow incommensurably large, such that these diagrams become incomprehensible. Adjacency matrices are a new, alternative way to visualize graphs. The visualization metaphor of matrix representations is in this thesis used to visualize a system's dependencies that exist on the file level. By using a hierarchical structure of a software system, it is possible to create surveyable matrices that visualize the relations that exist in a specific part of the system. By zooming in on a specific part of a matrix, a more detailed view of the corresponding part of the system is provided.

Van Ham created a tool that displays the call relations between the files of a software product by making use of this visualization metaphor ([1], [2]). Inspired by Van Ham's work, a web based tool was created for TIOBE Software, in which some essential conceptual changes were carried through. One change concerns arbitrary level nestings, which leaves the users more free on the definition on their systems' structures. The metaphor is strengthened by the matrices' independent axes, which enable users also to zoom in on columns and rows. Furthermore, the tool provides features that allow users to apply rules and to perform what-if scenarios on their systems and to introduce alternative system structures. The resulting tool is demonstrated for several of TIOBE Software's customers. These customers' own software systems were examined during these demonstrations. The tool is quite promising, so can be concluded from the evaluations of the given demonstrations.

Contents

Chapter 2

Introduction

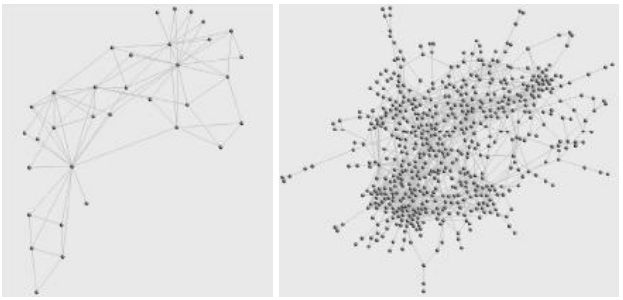
Computer systems are becoming increasingly complex due to both the growing number of users and the demand for more functionality. Besides, processors become more powerful, memory systems grow larger, operating systems provide more functionality and networks become faster. This growing complexity increases the difficulty of the tasks that developers are facing in the design and implementation of new technologies. To cope with this problem the system's architects need to have insight in their systems.

One way to gain insight in a complex system is by means of information visualization. Information visualization tries to exploit "the dynamic, interactive, inexpensive medium of graphical computers to devise external aids enhancing cognitive abilities"[3]. The visual artifacts that are brought forth by information visualization are designed to have "profound effects on people's abilities to assimilate information, to compute with it, to understand it, to create new knowledge"[3]. Some important aspects of information visualization are the interactivity and dynamics of the visual representation. Strong visual metaphors can enable the user to modify the visualization in real-time, which affords an unparalleled perception of the patterns and the structural relations that exist in the abstract data in question.

Software visualization is a branch of information visualization. It can give insight in three basic aspects of a software system, namely its structure, its runtime behavior and the code itself. Based on this insight, errors can be detected both earlier and easier, which means that not only the production time of the system decreases, but also that its quality improves. Nowadays, companies use tools that can be used to improve their software products. Some of these products are software visualization tools. This need for appropriate software visualization tools forms the foundation of this thesis.

2.1 Background

Dr. Ir. Frank van Ham worked on different software visualization tools during his PhD research. In 2000 van Ham started his PhD research on graph visualization at the Technische Universiteit Eindhoven. His supervisor was prof.dr.ir. Jarke J. van Wijk. Graph representations are a useful method to convey large amounts of information. However, it is difficult for humans to reason about a graph without a visual representation of that graph. Most often, graphs are represented by node-link diagrams: vertices are shown as nodes, edges as lines that connect the nodes.



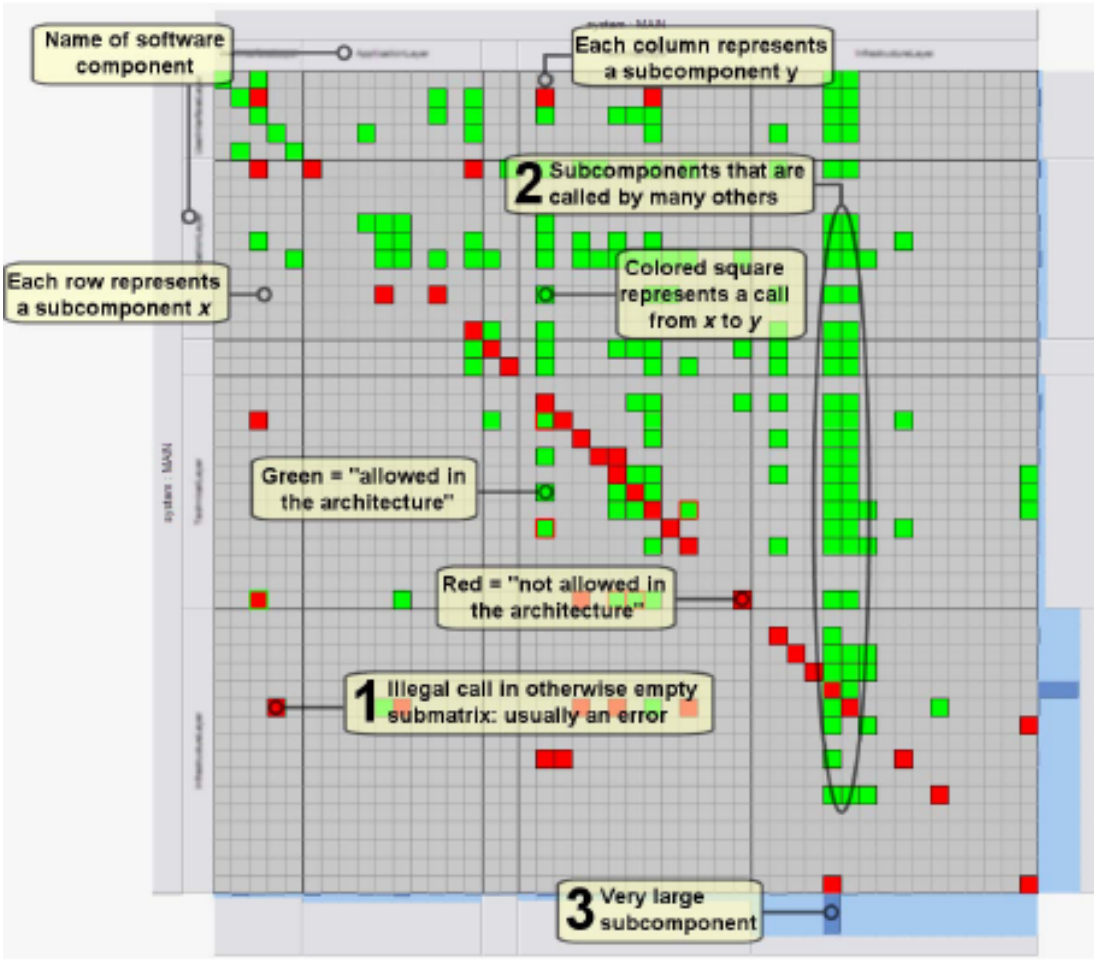
Two examples of node-link diagrams. The smaller graph contains 34 vertices, where the larger (but by no means large) graph contains 824 vertices [1].

As can be concluded from the above two figures, node-link representations of graphs do not always increase the users' cognition of the graphs. Van Ham stated his research question as follows: "How can we use computer supported visualization to help users gain insight in the structure of large graphs, with thousands or millions of nodes?". Van Ham observes that there exist a "small number of methods, techniques or tools that are able to visualize large graphs, so there are almost no best-practices known". Therefore, the nature of his thesis is exploratory.

Usually, node-link diagrams are used to visualize the dependencies in a software system. However, systems can grow incommensurably large, such that these diagrams become incomprehensible. In 2004 Van Ham worked on a tool called Matrixzoom. This tool visualizes large graphs in an alternative way by using matrix representations. Van Ham used his tool to visualize a system's cohesion based on its files and their interrelations, i.e., their *uses* relations. The figure on the next page illustrates the use and purpose of Matrixzoom.

On a smaller scale, adjacency matrices have been used to display dynamic properties of object oriented programs. Van Ham experienced that they exhibit a number of properties that make them attractive for the visualization of general graphs. He showed this by applying the matrix oriented visualization to the navigation of graphs with millions of edges. He had the disposition over two different datasets to test his tool. One dataset was formed out of the information retrieved from the SEER cancer database, where the other dataset came from a project of Philips Medical Systems. Van Ham concluded the following on the visualization metaphor:

- By making use of the recursive structure of adjacency matrices, it becomes more simple to employ smooth semantic zoom effects between abstraction levels. This zooming makes it easier for users to maintain context.
- The visual stability of the adjacency matrices provides a consistent bird-eye view of the entire graph.
- Matrix representations can serve as a useful externalization of possibly fragmented knowledge of the user.
- The fact that the graphical representation of the matrices can be computed in lineair time, makes them scalable, which does not hold for most algorithms that lay-out node-link diagrams.



An image generated by Matrixzoom. The figure also contains information on what is shown[1].

A user of Matrixzoom can draw conclusions based on the following:

- The green cells indicate the presence of a set of call relations between the corresponding components. This set of relations is allowed according to the design.
- The red cells indicate existing call relations that are forbidden according to the design.
- The red cells in an otherwise empty part of the matrix usually indicate programming errors that cause unwanted dependencies in the system.
- Long vertical columns of colored cells denote interface components. These are components that handle many incoming calls from other parts of the system.
- The bar diagram below the matrix indicates the size of each (sub)component. By this, users can immediately spot a component that is disproportionately large.
- Because of the large number of matrices that can be viewed by the users, it is easy for them to lose context. Matrixzoom provides an iconic representation of the system's hierarchy that decreases that chance.

Van Ham wrote a publication and held several colloquia on this visualization metaphor. This is how Drs. Paul Jansen, managing director of TIOBE Software, came in contact with this concept of software visualization.

TIOBE Software is specialized in assessing and tracking the quality of software. TIOBE Software measures the quality of a software system by applying widely accepted coding standards to it. Static analysis techniques are used to automate this process. Their solution consists of checking compliance to coding standards by means of code checkers, monitoring coding standard compliance through time by means of so called quality databases and by improving programs with the aid of code beautifiers. The result is a quality system that monitors software real-time.

TIOBE Software has insight in the software projects of many prominent companies, by which they are aware of the bottlenecks in large software projects. One of these bottlenecks is caused by the difference between the contrived design and the actual implementation of the system. Industry is very interested in tools that can be used to retrieve these differences.

Charmed by Matrixzoom's looks and potential, Jansen initiated a cooperation with the section of visualization of the department of computer science at the Technische Universiteit Eindhoven. Jansen and Van Wijk defined a master's project to implement and extend the concepts of Matrixzoom for TIOBE Software.

2.2 Objective

The objective of this project is to design and implement a tool that provides system architects a visual representation of their systems' structures and behavior in an effective and efficient way. The tool should give architects more insight in the relations that exist on the file level. Preferably, the tool should have the same look and feel as Matrixzoom, which does not meet TIOBE Software's requirements that concern performance and usage via the web. The choice was made to create a new tool, instead of modifying Matrixzoom, because this would be a too time-consuming task.

The visualization metaphor was demonstrated for TIOBE Software's customers. A number of functional requirements arose from these demonstrations and were subsequently claimed, next to TIOBE Software's, mostly non-functional, requirements. These requirements specify how and for what purpose the visualization metaphor of matrix representations should be used. A proper design should be created based on these requirements. Subsequently, the tool should be created in accordance with this design.

For future work, an elaboration should be made of other possible features that are longed for by industry, but could not be implemented within the duration of the project.

2.3 Outline of this thesis

In chapter 3 the requirements for the tool are listed. These requirements are divided into a set of functional and a set of non-functional requirements. The chapter also discusses Matrixzoom's shortcomings with respect to these requirements.

The required knowledge to understand the context and features of the tool is presented in chapter 4. It works out some notions and definitions that are frequently used in this thesis.

Chapter 5 is an elaboration of the visualization metaphor. It describes what the matrices imply and how they are constructed. Besides, it handles the usage and benefits of the zooming actions within the matrices. The chapter also states the differences between Matrixzoom and the tool described in this thesis, with respect to the interpretation and implementation of the metaphor.

Chapters 6 and 7 handle two different aspects that play an important role regarding the functionality of the tool. Chapter 6 explains the notion of rules and their characteristics. Chapter 7 declares the application and prescriptions of structure manipulations. These chapters deal with the aspects that bring about the most important features of the tool. Chapter 8 handles other relevant features that came up during the course of the project and improve the usage of the tool in different ways.

Chapters 9, 10 and 11 discuss the practical side of the tool. Chapter 9 handles the design decisions that were made during the project. Among other things, it indicates that the functionality of the tool is divided over different parts. In chapter 10, the functionality of these different parts are discussed in terms of the tool's graphical user interface. Chapter 11 deals with the implementation according to the design. It mostly handles the non-trivial implementation issues.

Chapter 12 is an evaluation of the tool. It handles the course of the given demonstrations, which is followed by an elaboration of the functionalities that could not be implemented during the project due to lack of time.

Chapter 13 contains the conclusion of the thesis.

The conclusion is followed by a listing of all resources that were consulted during the project.

Chapter 3

Requirements

Because of the positive results of Van Ham's research on matrix representations, TIOBE Software wanted to use his tool as a reference application for their own tool. However, for practical usage, some non-functional requirements were imposed, which caused that the design and implementation of the visualization method would not be as straightforward as a copy-paste job. Next to these non-functional requirements some functional requirements were claimed, which came from some of TIOBE Software's customers and arose after a demonstration of Matrixzoom. This chapter is an elaboration of the imposed requirements.

3.1 Functional requirements

- **[FR1] Exploring the file relations**
  Most architects want to make observations on the current state of their software systems. By using the tool, the architects should be enabled to view the file relations that exist in their current systems. Through this, they can determine their system's modularity and make assumptions and draw conclusions on the structure and behavior of their software systems. The tool should initially be used to visualize a system's include relations (more on this in section 4.4).

- **[FR2] Check for architectural violations**
  Architects follow and implement certain architectural rules when designing a system. The tool should be able to check whether the implemented system actually follows these predefined architectural rules. The relations that violate these rules should consequently be marked as forbidden.

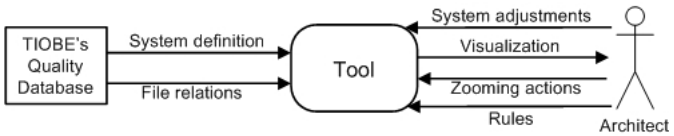
- **[FR3] Setting rules**
  The design document defines the software system's components and their interrelations. Most probably, some components are not allowed to interrelate. These unwanted relations can be considered as structural rules that should

apply to the system. Architects must be enabled to declare these rules in the tool and consequently view the file relations that violate them.

- **[FR4] Spotting unwanted relations**
  Architects want to find unwanted file relations in their systems. These unwanted file relations are those file relations that should not be there according to the design documents, but do exist in the current system. Therefore, they are considered to be forbidden. If an architect considers a file relation to be erroneous, then the tool should enable him to mark that file relation as forbidden.

- **[FR5] Decreasing the amount of violations**
  If architects are enabled to declare rules and mark file relations as forbidden, then it is possible that violations on these regulations exist in a software system. The lower the amount of violations, the more the system is in accordance with its devised design. The architects want to use the tool to foresee how the relations, and so the violations, manifest themselves when the structure of a system is altered. If some changes result in a lower amount of violations, then the architects can decide to implement these changes. This way, the architects can gradually improve their systems.

The following diagram represents the context of the tool:



A diagram that represents the context of the tool.

The diagram can be split in three different parts. The leftmost entity resembles TIOBE Software's Quality Database that contains all sorts of information about the systems that are inspected by TIOBE Software's framework. The tool that is described in this thesis is represented in the middle of the diagram. The rightmost part is the user of the tool, i.e., the architect.

The arrows that connect the different entities represent the information flows between the entities. The Quality Database provides the needed information for the tool, which consists of a system definition and the set of file relations.

The tool provides the architect with a visualization of his system's include relations by means of matrix representations. The architect can get a finer grained view on the include relations at a lower level of abstraction by means of zooming actions within a matrix. Furthermore, the architect is enabled to define rules and manipulate the system.

The following chapters are elaborations of the different entities and information flows that are part of the diagram.

3.2 Non-functional requirements

- **[NFR1] - Web enabled**
  The tool should be accessible through a web browser. Matrixzoom is a stand-alone application that requires installation on a user's computer. TIOBE Software stated this requirement for two reasons. The first reason concerns the large number of expected users that work on different computers. All these users should be enabled to make use of the tool, independent of their working places. Access via web browsers is a good solution for this, because most computers have web browsers installed. Secondly, TIOBE Software's tool set comes as a web application and it would be very desirable if this new tool could eventually be integrated into this framework.
- **[NFR2] - No other installation required**
  Another non-functional requirement, which has almost the same reasoning as **NFR1**, claims that the tool should not require an installation of any other piece of software besides the browser on the client. Many tools require the installation of products like Java Runtime Environment (JRE), Macromedia Flash, etcetera. TIOBE Software wants to keep the installation of their products as simple as possible by making their scripts independent of browsers, plug-ins and other software. Besides, it prevents problems concerning the versions of these products.
- **[NFR3] - Performance**
  The touch and feel of the tool should be like in Matrixzoom, where the zooming is smooth and understandable. However, making such animations web based is quite tricky, because browsers do not come with drawing libraries such that they are able to create these animations on their own. The leading criterion on the animations is that a fifty by fifty matrix, with a hundred cells filled, should zoom in smoothly to a matrix with the same properties. Next to this, the animation should be initiated within two seconds. Matrices larger than this one are allowed to run more bumpy or take longer to initiate.

3.3 Matrixzoom's shortcomings

Van Ham's Matrixzoom, does not satisfy the imposed requirements, because it

- is not web-enabled
- needs to be installed on the client
- only enables users to define very basic rules
- can not be used to manipulate a system and observe the results
- employs strict requirements on a system's structure

The first two shortcomings are the major reasons to design and create a whole new tool, instead of adjusting Van Ham's prototype.

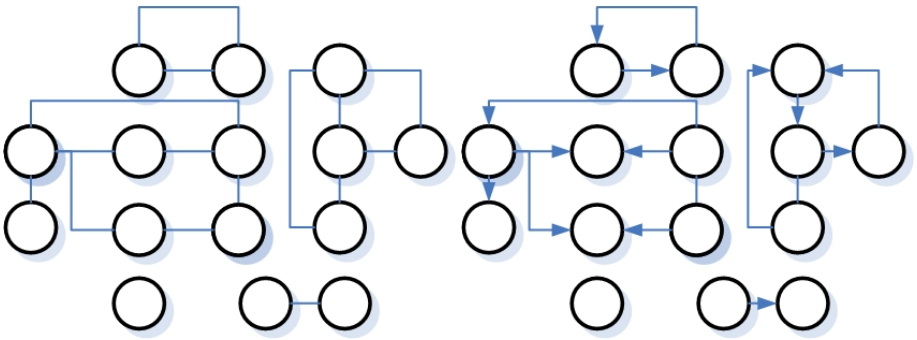
Chapter 4

Data analysis

This chapter handles notions and definitions that are frequently used in this thesis and that clarify the purpose and the application of the tool.

4.1 Graphs

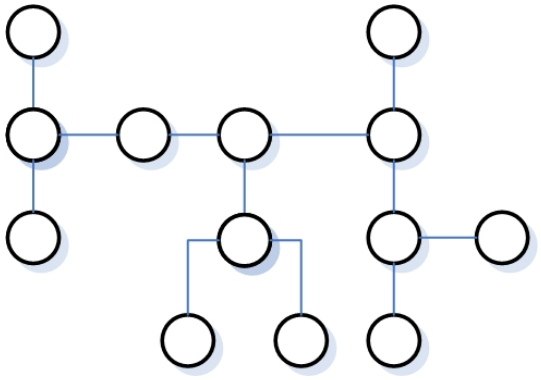
A graph $G = (V, E)$, in the mathematical sense, consists of a set of vertices V that are connected by a set of edges E. A directed edge $e \in E$ that connects vertices $v_1, v_2 \in V$ will be denoted as (v_1, v_2). A directed graph G has its set of edges E defined as a subset of $V \times V$. An edge (v_1, v_2) represents a connection between the *tail* v_1 and the *head* v_2. That is, v_2 can be reached from v_1, but v_1 not from v_2.



The left diagram represents a normal graph, where the right diagram represents a directed graph.

4.1.1 Trees

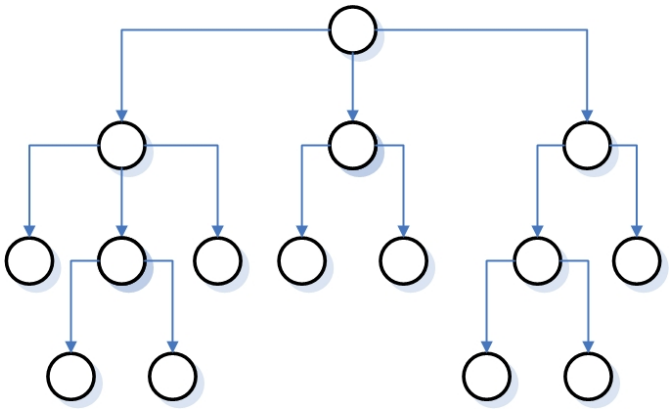
Among the various types of graphs are trees. A tree is a connected (non-directed) acyclic graph, which means that any vertex $v_1 \in V$ is connected with any other vertex $v_2 \in V \wedge v_1 \neq v_2$ by a number of edges $e \in E$ that form an acyclic path. $G\ is\ Tree \Rightarrow (\forall v_1, v_2 : v_1, v_2 \in V \wedge v_1 \neq v_2 : v_1 \twoheadrightarrow v_2) \wedge \neg(\exists v : v \in V : v \twoheadrightarrow v)$, where $v_1 \twoheadrightarrow v_2$ means that there exists a path along a set of vertices that is connected by a set of edges, such that this path starts with v_1 and ends with v_2.



A tree depicted as a node-link diagram.

4.1.2 Rooted trees

A rooted tree, also known as an hierarchical tree, is a directed graph based on a tree. In a rooted tree is exactly one vertex $v_1 \in V$, the root, for which there is a path from v_1 to all other vertices $v_2 \in V$. Besides, the in-degree of all vertices is at most one. $G\ is\ rooted\ tree \Rightarrow G\ is\ directed \wedge (\exists^1 v_1 : v_1 \in V : (\forall v_2 : v_2 \in V \wedge v_1 \neq v_2 : v_1 \twoheadrightarrow v_2)) \wedge (\forall v : v \in V : indegree(v) \leq 1)$



A rooted tree depicted as a node-link diagram.

Let $G = (V, E)$ be a rooted tree, then the following holds:

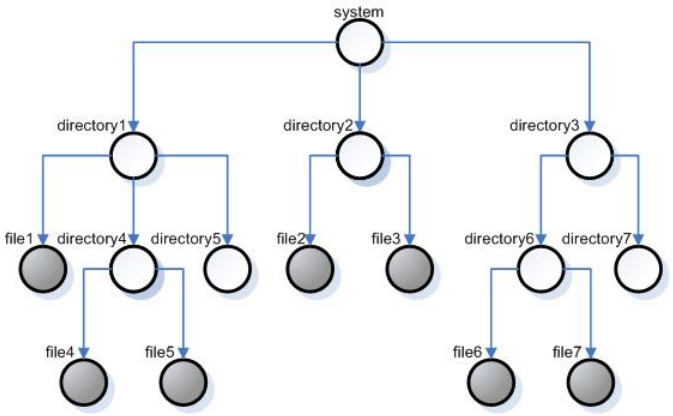
- **Child relations**
  The vertices adjacent to a given vertex $v_1 \in V$ are called its children ($v_2\ is\ child\ of\ v_1 \equiv v_2 \in V \wedge (\exists e : e \in E : e = (v_1, v_2))$).
- **Parent relations**
  The vertex adjacent from a given vertex $v_1 \in V$ is called its parent ($v_2\ is\ parent\ of\ v_1 \equiv v_2 \in V \wedge (\exists e : e \in E : e = (v_2, v_1))$).
- **Internal vertices**
  The internal vertices of a tree are those vertices with both a parent and children ($v_1\ is\ internal\ vertex\ of\ G \equiv v_1 \in V \wedge (\exists e_1, e_2, v_2, v_3 : e_1, e_2 \in E \wedge v_2, v_3 \in V : e_1 = (v_2, v_1) \wedge e_2 = (v_1, v_3))$).
- **Leaves**
  A vertex with no children is called a leaf ($v_1\ is\ leaf\ of\ G \equiv v_1 \in V \wedge \neg(\exists e : e \in E : e = (v_1, v_2))$).
- **Descendants**
  The transitive closure of the child relations of a given vertex $v_1 \in V$ is commonly denoted by the descendant relation. ($v_2\ is\ descendant\ of\ v_1 \equiv v_2 \in V \wedge v_1 \twoheadrightarrow v_2$)
- **Ancestors**
  The transitive closure of the parent relations of a given vertex $v_1 \in V$ is commonly denoted by the ancestor relation. ($v_2\ is\ ancestor\ of\ v_1 \equiv v_1 \in V \wedge v_2 \twoheadrightarrow v_1$).
- **Subtrees**
  A subtree formed by a vertex $v_1 \in V$ and its descendants is a rooted tree with v_1 as root. This subtree contains all vertices $v_2 \in V$ that are descendants of v_1 and it contains all edges $e \in E$ that have both ending points in this set. With $v_1 \in V$, $subtree(v_1) \equiv (\{v \in V | v_1 \twoheadrightarrow v \vee v_1 = v\}, \{e \in E | (\exists v_2, v_3 : v_2, v_3 \in V \wedge (v_1 \twoheadrightarrow v_2 \vee v_1 = v_2) \wedge (v_1 \twoheadrightarrow v_3 \vee v_1 = v_3) : e = (v_2, v_3))\})$.
- **Depth**
  The depth of a vertex $v_1 \in V$ is the length of the path from the root vertex to v_1 and is denoted as $d(v_1)$.

Trees are a nice kind of graphs, because they are easy to understand. Although trees can grow tremendously big, they still can be useful to make observations. Next, they are very easy to construct and they can be used to store all kinds of information. Another reason why trees are popular is because there exist dozens of algorithms that work on tree structures. Common uses for rooted trees are to:

- Manipulate hierarchical data
- Make information easy to search (Breadth First Search, Depth First Search, etc.)
- Manipulate sorted lists of data

4.1.3 The directory structure as a rooted tree

Software systems can be represented as rooted trees. For example, the directory structure of a system can be represented as a rooted tree. A directory structure is by definition hierarchical. The root vertex of the tree represents the root directory of the system. The set of internal vertices consists of the non-empty directories, where the set of leaves consists of the files and the empty directories of the system. The edges in the tree correspond to the contains/ischild relations of the system. People are very familiar with this type of system representations, because most operating systems represent the contents of a computer system as a rooted tree.



A small software system depicted as a node-link diagram. The grey vertices correspond to files of the system, where the white vertices are directories. The edges in the diagram represent the contains/ischild relation between the vertices.

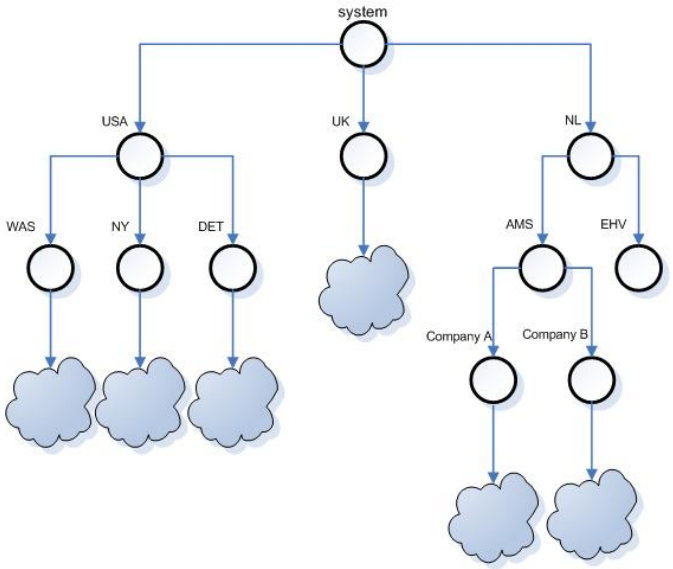
This type of graphs that represents a system's structure will from now on be referred to as G_T, which equals (V_T, E_T).

4.1.4 Other structures as rooted trees

The notion of directory structures is already used in previous sections. This structure is the most common structure of a software system, because it is usually easy to retrieve. However, companies may define other structures on their software systems, which may give a better overview of their systems. Examples of these structures are organizational structures, layered architectures and class diagrams.

A property of these structures is that they can also be represented by a rooted tree, like the directory structure. This is not a coincidence, because hierarchy is the best way to manage a system's complexity. These structures have one particular root, a set of internal vertices and a set of leaves. Some of these parts can be retraced from the directory structure, where other parts are specific for that structure. The edges in the tree are defined by the structure's architect.

For example, an organizational view's vertices with depth one could resemble different countries. Their children could possibly resemble cities, while their children represent different companies. The descendants of the companies are the directories and files that are produced by the corresponding companies.



A small software system depicted in its organizational view. Country, city and company names are depicted above the corresponding vertices. The clouds resemble the directories and files that were produced by their parent vertices.

One of the main shortcomings that TIOBE Software encountered when they used Matrixzoom, concerned the fixed depth of the tree that represents the systems' structures. Matrixzoom requires each system to be split up into the levels *System*, *Layer*, *Unit*, *Module* and *Class*. This demands from Matrixzoom's users to make a mapping from their system to these levels of abstraction. TIOBE Software desires a tool that is more generic and that does not impose any constraints on which structure to use for the visualization of the file relations. Because nothing sensible can be said about the division of files in any system's structure, the tool should be able to deal with arbitrary structures.

4.2 File relations and graphs

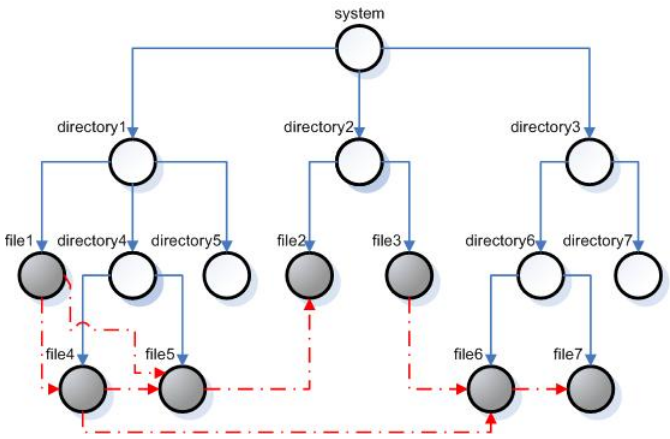
Next to this rooted tree that describes a system's structure, another graph exists that describes another part of the system, namely the graph that represents the relations that exist between the system's files. Examples of relations that exist between files are *uses*, *include*, *inheritance*, *containment* and *class reference* relations. The directed graph that represents the file relations of a system is referred to as $G_R = (V_R, E_R)$, where the set of vertices V_R is defined by the set of files and the set of edges E_R is instantiated by the relations that exist between the files. Note that $V_R \subseteq V_T$ holds.

4.3 Combining the two graphs

The two previous sections define two different types of graphs that can be used to represent different aspects of a software system. The first one defines the structure of a system by means of a tree G_T. This graph's set of edges E_T represent containment/ischild relations. The other graph, G_R, defines relations between the files of a system by its set of edges E_R. These two graphs can be combined in one larger graph, $G_C = (V_T, E_T \cup E_R)$. Thus, the resulting graph G_C represents both the system's structure and its relations that exist on file level.

The resulting graph is a general graph. The size of this graph, i.e., its number of vertices, can vary enormously. Its number of edges is arbitrary, because this strongly depends on the coherence of the system. The graph is locally clustered, because systems are implemented such that their modularity is as high as possible. Compared to other graphs, its diameter, i.e., the length of the longest path, is rather small. This is also caused by the way that software systems are constructed.

For a small software system, the resulting node-link diagram would look like the following diagram:



Between the files is a set of relations, represented by the dashed lines. The other lines between the vertices define the structure of the system.

Architects of this system can use this diagram to make assumptions and draw conclusions on their system, because it is rather conveniently arranged. However, as the number of vertices and edges increases, this type of diagrams becomes more complex. A node-link diagram is a respectable solution when it contains less than a hundred vertices, but it becomes impossible to observe a system's structure and behavior as the diagram grows larger. The visualization metaphor used by Van Ham is another visualization method to visualize a combination of these two graphs in a more scalable and conveniently arranged way.

4.4 Include relations

Visualizing the relations between a system's files by using matrix representations is seen as attractive. This is one of the conclusions of Van Ham. That is why the main part of the tool-to-be-built should be using this strong metaphor. Matrix-zoom is used to visualize the *uses* relations of a system. However, the visualization metaphor can be used to visualize all types of file relations. It is TIOBE Software's wish to visualize the include relations between a system's files.

Include relations are one of the relations that can exist between files. They are one of the major indicators of software modularity. Includes allow programmers to use the contents of a file in another file. Programmers can group declarations of common functions and code into a file and then just make a reference to, i.e., an include of, that file in all the files that need to use those functions. So, rather than having to rewrite functions, a programmer can write them once and access them easily from all the files that need them.

Unwanted include relations can possibly lead to an erroneous system, so it can be considered worthwhile to check for them.

4.5 Transitive relations

If a file $v_1 \in V_T$ includes a file $v_2 \in V_T$ that includes again a file $v_3 \in V_T$, then v_1 can make use of the contents of v_3. This property of include relations brings about the notion of indirect relations; the so-called transitive relations. Van Ham also worked on the visualization of indirect relations. He argues that indirect dependencies are hard to detect in a matrix visualization, since relations that are semantically close (like $e1, e2 \in E_R$, for which holds that $e1 = (v_1, v_2)$ and $e2 = (v_2, v_3)$, with $v_1, v_2, v_3 \in V_T$) can visually be located far apart. Van Ham solved this partially in two ways, namely by indicating the neighbourhood of a relation upon selection and by displaying the n-step closure of a relation. He found out that both are not satisfying solutions. Nevertheless, the industry wants to make assumptions and draw conclusions on their systems based on both the direct and indirect relations that exist in their systems. The indirect relations should therefore be calculated and added to the set E_R. Consequently, these relations should also be visualized in the matrices.

After the calculation of all indirect relations, it is possible that more relations from $v_1 \in V_T$ to $v_2 \in V_T$ exist in the set of direct and indirect relations. In order to ease calculations, all these relations are removed from the set of relations, except for the relation with the shortest path from v_1 to v_2. This way, the maximum number of relations $e \in E_R$, $e = (v_1, v_2)$, with $v_1, v_2 \in V_T$ equals one.

4.6 The initial data of the tool

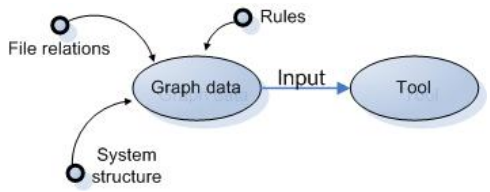
For all customers, TIOBE Software stores the include relations in their so called quality database. This is done to enable the analysis of the header files of a software project. TIOBE Software also stores absolute pathnames. Using these pathnames, it is possible to reconstruct the system's directory structure. This means that both the graph that represents the include relations and the tree that represents the system's directory structure are present for the visualization of the include relations of TIOBE Software's customers' systems.

Chapter 5

The visualization metaphor

Adjacency matrices are an alternative way to visualize graphs. This visualization metaphor can much easier cope with large graphs, because it is not necessary to compute a good node-link layout. In his research, Van Ham uses matrix representations of a graph to visualize relations between components in large software systems. This chapter is an elaboration of this visualization metaphor, which concerns the requirements **FR1** and **FR4**.

The graphs that are visualized using matrix representations are a combination of the tree that defines the system's structure, the graph that represents the existing file relations and the graph that embodies the declared structural rules (more on rules in chapter 6).



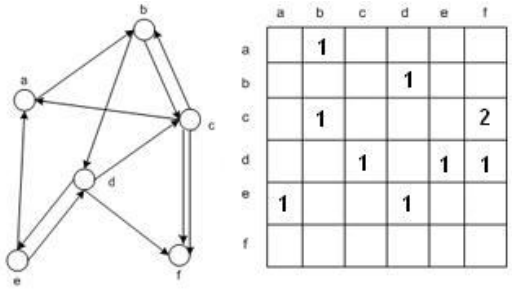
A representation of the datamodel of the input-graph.

5.1 The visualization metaphor

This thesis focusses on matrix representations that are used for visualizing include relations that exist between files of a software system. The metaphor does not use one big matrix to show these relations, but multiple smaller matrix representations, which are constructed by making use of the software system's structure. In this section is explained how these smaller matrices are created and how these are linked together.

5.1.1 The construction of a matrix

In accordance with the structure of a software system, the visualization metaphor uses abstractions to visualize the matrix representations. These matrices are adjacency matrices that visualize the existence of file relations in a system. The adjacency matrix of a finite graph $G = (V, E)$ is a $|V| \times |V|$ matrix where a cell a_{ij} represents the number of edges from vertex i to vertex j.



An example of an adjacency matrix. Left is a general graph and right is the corresponding adjacency matrix.

An adjacency matrix defines the same graph as the corresponding node-link diagram, because the graph's vertices are presented in the rows and columns, where its edges are represented by the numbers in the cells. However, such an adjacency matrix takes relatively much space, compared to a node-link diagram. It is possible to give a better overview of the graph by making use of the tree that defines a system's structure hierarchically.

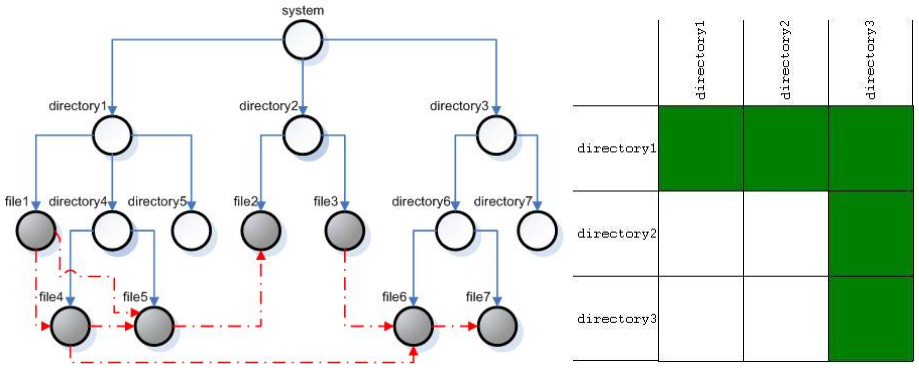
For every combination of two non-leaves exists a matrix representation that visualizes the relations that exist in the enclosed part of the corresponding system abstraction. The first vertex represents the focus on the y-axis of the matrix, while the other vertex represents the focus on the x-axis. These vertices are called the focus vertices. The matrix' rows and columns are formed by the children of the focus vertices (which explains why there is no matrix representation if one of the focus vertices is a leaf).

Van Ham uses another approach where the rows and columns are formed by the grandchildren (the children of the children) of the focus vertices. Although his approach gives the users more insight in the structure and contents of the components, the number of rows and columns of the matrices is much higher, which causes the user's cognition of the file relations to be much less. In this project the visualization metaphor is used for the latter goal, so the choice was made to construct the matrices as described above.

The cells of the adjacency matrices are filled if and only if there exist one or more relations from a file in the subtree formed by the row's corresponding vertex and its descendants to a file in the subtree formed by the columns's corresponding vertex and its descendants.

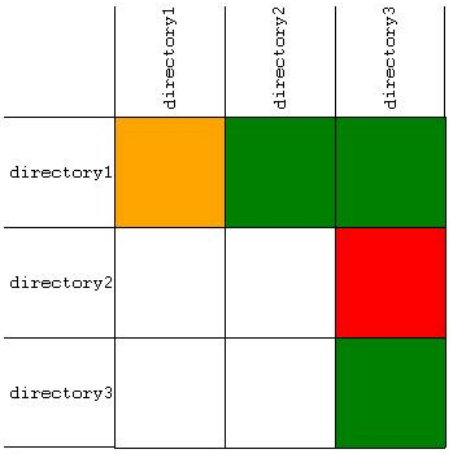
More formally, let the system be defined by the graph $G_C = (V_T, E_T \cup E_R)$. Next, let the focus be on (v_1, v_2), with $v_1, v_2 \in V_T$ and both non-leaves, then the following holds for the corresponding matrix:

- The rows are formed by the set of children of v_1 ($\{v \in V_T | v \; is \; child \; of \; v_1\}$)
- The columns are formed by the set of children of v_2 ($\{v \in V_T | v \; is \; child \; of \; v_2\}$)
- A cell (v_3, v_4) of the matrix is filled $\equiv (\exists v_5, v_6, e : v_5, v_6 \in V_T \wedge e \in E_R \wedge e = (v_5, v_6) : v_3 \twoheadrightarrow v_5 \wedge v_4 \twoheadrightarrow v_6)$.



Left is an example of a software system and its file relations. The focus nodes are both on the root vertex of the system. Right is the corresponding matrix representation.

The coloring of a cell of a matrix is deduced from Matrixzoom, which uses red to visualize forbidden relations and green for allowed relations. The tool described in this thesis uses three colors to fill a cell. The possible colors are green, red and orange; the colors of a traffic light. If all involved file relations are allowed, then the cell is colored green, while it is colored red if all these relations are forbidden. If both allowed and forbidden relations exist in the corresponding system abstraction, then the cell will be colored orange.



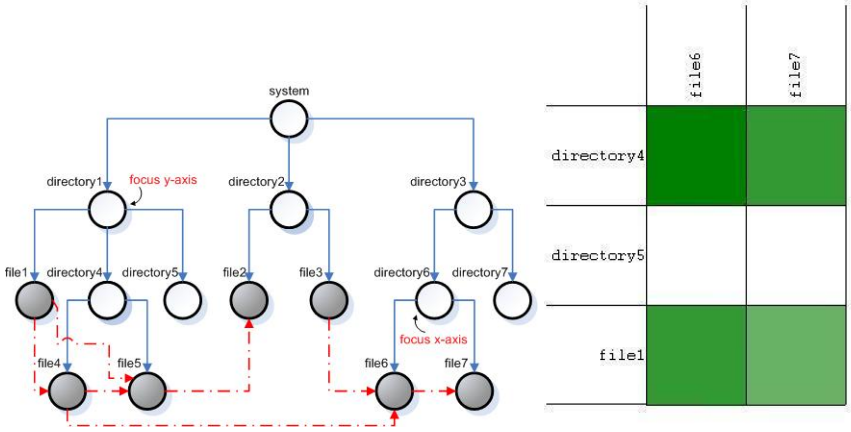
This diagram shows a matrix representation of the same system on which now some rules have been defined, such that it contains some forbidden relations. The focus nodes are both on the root vertex of the system.

Transitive relations

The importance of indirect relations is considered to be much less than that of the direct relations. Therefore, two different features are needed to emphasize the difference. Firstly, showing indirect relations should be an optional feature, such that users are able to leave out the indirect relations from the matrices if they are not interested in those. Secondly, color should be used to show the difference between these two types of relations. With the presence of indirect relations there are four different scenarios for a matrix' cell. The following relations can be present between the cell's corresponding subtrees:

- no direct nor indirect relations
- a set of direct relations, but no indirect relations
- no direct relations, but a set of indirect relations
- both a set of direct and a set of indirect relations

Different saturations can be used to show the difference between these scenarios. In the first case, the cell should be uncolored. If the corresponding system abstraction only contains allowed relations, then the cell should be colored dark green in the second and last case, where the cell should be colored lighter green in the third case. If it also contains forbidden relations, then it is colored orange. The saturation of a cell depends on the depths of the enclosed relations. The depth of a direct relation is considered one. The depth of an indirect relation corresponds to the number of direct relations that bring about the indirect relation. The saturation of a cell lessens as the minimum depth of the set of enclosed relations increases.



Left is an example of a software system and its file relations. The focus nodes are pointed out in the figure. Right is the corresponding matrix representation. The figure shows the different colorings of the cells.

A transitive relation is considered not to be affected by rules, such that the red and orange cells in a matrix correspond to the existence of a set of forbidden direct relations. This way, the direct relations that need attention come out more clear. Consider a direct relation from A to B and one from B to C, then there is a transitive relation from A to C. If a rule states that relations from A to C are forbidden, then yet this transitive relation is considered allowed. In this case, a good architecture will define a rule that forbids at least one of the involved direct relations.

5.1.2 Zooming on abstractions

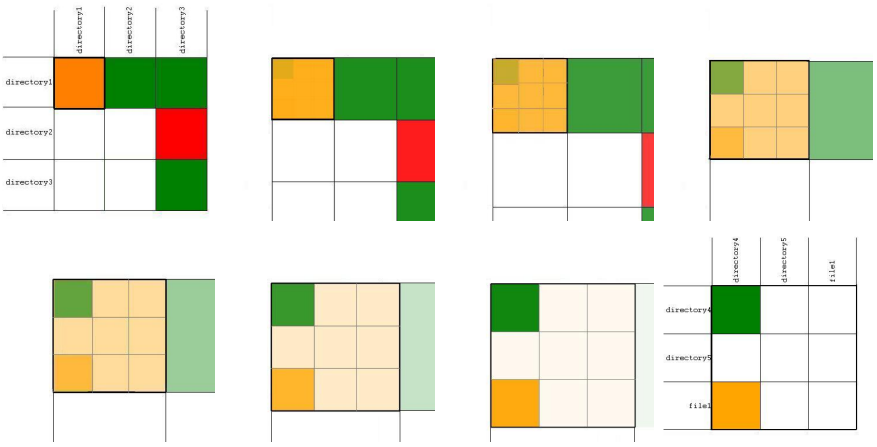
The visualization metaphor is strengthened by the fact that the matrix representations allow zooming actions. This semantic zooming allows users to behold all different abstractions of their systems, which gives the impression of a single coherent information space. A zoom-in action on a cell (v_1, v_2) causes the focus vertices to change into v_1 and v_2 themselves. A zoom-out action on a cell (v_1, v_2) causes the focus vertices to change into their own parents. These zoom actions bring about a smooth transaction that starts with the current matrix representation and ends in the state where the new matrix is shown. The figures on the following page are screen shots taken during a zoom in action.

Because of performance reasons, zooming in on an uncolored cell is disabled. Besides, considering the purpose of the tool, it would be quite useless, because the underlying matrix does not encapsulate any relations and it increases the chance of users getting lost in the maze of matrices. Zooming in on a colored cell is only possible if an underlying matrix exists, which means that it is not possible to zoom in on a cell that is in a row or column that corresponds to a leaf vertex.

It is also possible to perform zooming actions on rows or columns, which allows the users to move their focus more freely among the vertices of the tree that represents the system's structure. Matrixzoom does not allow this. Zooming in on rows or

columns is only allowed if they contain colored cells and do not correspond to leaf vertices, for the same reasons as mentioned for zooming on uncolored cells. Zooming in on a row or column results in a change of the matching focus vertex in the corresponding vertex of the selected row or column.

Zooming out on a cell of an arbitrary matrix is always possible as long as both focus vertices are not the root vertex. Zooming out on a column or row is also possible as long as the matching vertex is not the structure's root vertex.



Eight figures that show the zoom in action on the cell in the upper left corner of the matrix.

If the focus vertices are (v_1, v_2) and the system is described by $G_C = (V_T, E_T \cup E_R)$, then the following holds for this visualization metaphor:

- **Zooming in on a cell** (v_3, v_4)
  $\{pre : v_3, v_4 \in V_T \wedge (\exists e_1, e_2 : e_1, e_2 \in E_T : e_1 = (v_1, v_3) \wedge e_2 = (v_2, v_4)) \wedge (v_3, v_4)\ is\ filled\}$
  zoominoncell(v_3, v_4);
  $\{post : focus = (v_3, v_4)\}$

- **Zooming in on a row** (v_3)
  $\{pre : v_3 \in V_T \wedge (\exists e : e \in E_T : e = (v_1, v_3)) \wedge (\exists v_4, e : v_4 \in V_T \wedge e \in E_T : e = (v_2, v_4) \wedge (v_3, v_4)\ is\ filled)\}$
  zoominonrow(v_3);
  $\{post : focus = (v_3, v_2)\}$

- **Zooming in on a column** (v_4)
  $\{pre : v_4 \in V_T \wedge (\exists e : e \in E_T : e = (v_2, v_4)) \wedge (\exists v_3, e : v_3 \in V_T \wedge e \in E_T : e = (v_1, v_3) \wedge (v_3, v_4)\ is\ filled)\}$
  zoominoncol(v_4);
  $\{post : focus = (v_1, v_4)\}$

- **Zooming out on a cell** (v_3, v_4)
  $\{pre : v_3, v_4 \in V_T \wedge (\exists e_1, e_2 : e_1, e_2 \in E_T : e_1 = (v_1, v_3) \wedge e_2 = (v_2, v_4)) \wedge (\exists v_5, v_6, e_1, e_2 : v_5, v_6 \in V_T \wedge e_1, e_2 \in E_T : e_1 = (v_5, v_1) \wedge e_2 = (v_6, v_2))\}$
  zoomoutoncell(v_3, v_4);
  $\{post : focus = \{(v_5, v_6) | v_5, v_6 \in V_T \wedge (\exists e_1, e_2 : e_1, e_2 \in E_T : e_1 = (v_5, v_1) \wedge e_2 = (v_6, v_2))\}$

- **Zooming out on a row** (v_3)
  $\{pre : v_3 \in V_T \wedge (\exists e : e \in E_T : e = (v_1, v_3)) \wedge (\exists v_5, e : v_5 \in V_T \wedge e \in E_T : e = (v_5, v_1))\}$
  zoomoutonrow(v_3);
  $\{post : focus = \{(v_5, v_2) | v_5 \in V_T \wedge (\exists e : e \in E_T : e = (v_5, v_1))\}$

- **Zooming out on a column** (v_4)
  $\{pre : v_4 \in V_T \wedge (\exists e : e \in E_T : e = (v_2, v_4)) \wedge (\exists v_6, e : v_6 \in V_T \wedge e \in E_T : e = (v_6, v_2))\}$
  zoomoutoncol(v_4);
  $\{post : focus = \{(v_1, v_6) | v_6 \in V_T \wedge (\exists e : e \in E_T : e = (v_6, v_2))\}$

5.2 Advantages and disadvantages

The advantages of the visualization metaphor of matrix representations are the following:

- The visual stability of the adjacency matrices provides a consistent bird-eye view of the entire graph.
- Matrix representations can serve as a useful externalization of possibly fragmented knowledge of the user.
- The fact that the graphical representation of a matrix can be computed in line-air time, makes them also scalable, which does not hold for most algorithms that lay-out node-link diagrams.
- By making use of the recursive structure of adjacency matrices, it becomes more simple to employ smooth semantic zoom effects between abstraction levels. This zooming makes it easier for users to maintain context.
- The metaphor has a recursive structure. This means that the visual representation of all relations between the descendant vertices of $v_1, v_2 \in V_T$ are by definition contained in the visual representation of the aggregated relation between v_1 and v_2. This makes it easy to construct multiscale visualizations. This means that a specific file relation can only be found by zooming in on the cells or rows and columns that belong to the involved files' ancestors.

- The metaphor is stable and predictable. In contrast to other popular graph visualization methods, the addition of a file relation is guaranteed to never radically change the layout of a matrix representation, since every relation has its own pre-reserved section of visualization space.
- The metaphor offers space to display additional information. In a standard node-link diagram, the edges are usually displayed as thin lines of varying length and orientation. This makes picking, coloring and labeling more difficult. In contrast, a matrix representation contains one distinct space per relation and the four sides of the matrix can be used to display vertex attributes. Two of these four sides are used to display vertex names. The other two sides can be used to visualize vertex attributes such as subsystem size, the number of changes or a number of system metrics such as (inverse) coupling, complexity etc.

However, the metaphor also has disadvantages:

- The 'semantic closeness equals visual closeness' principle does not hold for this metaphor. A relation $e_1 \in E_R$ from $v_1 \in V_T$ to $v_2 \in V_T$ is semantically close to a relation $e_2 \in E_R$ from v_2 to $v_3 \in V_T$. However, in the matrix representations, these two relations can be visually far apart.
- Either the size of a matrix cell or the size of the whole matrix depends on the current view. As the view changes to another matrix, one of these sizes can change drastically, depending on the number of rows and columns that are shown. These changes lead to a poorer understanding of what is visualized.
- If the current view is a one by one matrix, then a zoom action to a following view does not look like a zoom action, but more like a fade in - fade out action.

Chapter 6

Rules

Chapter 3 states the requirements for the tool. A number of these requirements contain the terms *forbidden*, *violations* and *rules*. Design rules produce forbidden relations and a forbidden relation in a system causes the presence of a design violation in the system. This chapter is an elaboration of the notion of rules and it exemplifies how these can be applied.

6.1 Atomic rules

The following function is defined for each rule: $R(e) = (c, a)$, where e is the input relation $e \in E_R$ from $v1$ to $v2$, with $v1, v2 \in V_T$. c Is $true$ if and only if the relation is in the *scope* of the rule, i.e., if the rule is applicable for this edge. The value of a depends on the effect of the rule. a Equals $false$ if the relation is forbidden by the rule and equals $true$ otherwise. The result of the function is a tuple, such that the problems that arise when combining multiple rules can be tackled. More on this later in this chapter.

Two different types of rules can be applied to a system, namely structural and architectural rules. These two types of rules are elaborated in the following subsections.

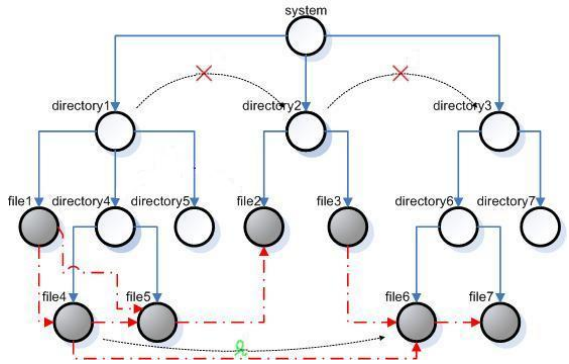
6.1.1 Structural rules

A system's design points out which parts of a system are allowed to interrelate with each other and which are not. This can be on a high level of abstraction, like component level, but also on file level. These design rules can be defined as relations between the parts, which means that the rules can be declared on the vertices V_T of the graph that represents the system's structure. Hence, a structural rule is defined from one vertex $v_1 \in V_T$ to another vertex $v_2 \in V_T$. The rules form the set of edges E_S and are referred to as the system's structural rules.

A rule defined from a vertex $v_1 \in V_T$ to a vertex $v_2 \in V_T$ affects all relations $e \in E_R$ that are from a file in the subtree formed by v_1 and its descendants to a file in the subtree formed by v_2 and its descendants. These relations fall within the *scope* of the structural rule. A rule can cause all affected file relations to be forbidden, but can also cause all effected file relations to be allowed. The effect of a rule is determined by the architect and it is stored in a boolean attribute of the corresponding edge $e \in E_S$. Rules that cause file relations to be forbidden are referred to as forbidden rules, where a rule that allows the relations within its scope is called an allowed rule.

Recall the function $R(e)$. For a file relation $e_1 \in E_R$, where $e_1 = (v_1, v_2)$, with $v_1, v_2 \in V_T$ and a structural rule $e_2 \in E_S$, where $e_2 = (v_3, v_4)$, with $v_3, v_4 \in V_T$ the following holds: $c \equiv (v_1 \twoheadrightarrow v_3 \wedge v_2 \twoheadrightarrow v_4)$, which means that c equals $true$ if and only if the relation is in the scope of the rule and $false$ otherwise. If the rule is an allowed rule, then a equals $true$ for all relations. If the rule is a forbidden rule, then a equals $false$ for all relations where c equals true, while it equals $true$ otherwise.

The following diagram shows the graph $G_{TOT} = (V_T, E_T \cup E_R \cup E_S)$, which represents a small system's structure, its file relations and several structural rules that are defined on the structure.



Next to the file relations, there are small-dashed arrows that represent the rules between components. A cross in such an arrow indicates a forbidden rule, where a curlicue stands for an allowed rule. These rules cause some of the relations to be forbidden.

6.1.2 Architectural rules

Architectural rules are another type of rules. Architectural rules, defined by the software architects, are a set of rules or guidelines that need to be followed during the design, coding and implementation of a system. An example of an architectural rule is one that forbids a layered module to use another module that is positioned at a higher layer. Another example is one that forbids *uses* relations to be cyclic. This rule forbids a module A to use module B if module B already uses module A. The relations that do not follow an architectural rule are considered to be forbidden.

Unfortunately, an architectural rule can not be defined as a single edge between two vertices of the tree that defines a system's structure. Therefore, architectural rules are not as generic as structural rules.

For architectural rules, the return value of c in the function $R(e)$ equals $true$ if the relation violates the architectural rule and equals $false$ otherwise. Note that $c \neq a$ holds for architectural rules.

After a study on include relations, three different architectural rules were found that deal with these relations. These architectural rules concern:

- Cyclic relations
- Relations that skip a level downwards
- Relations that go upwards

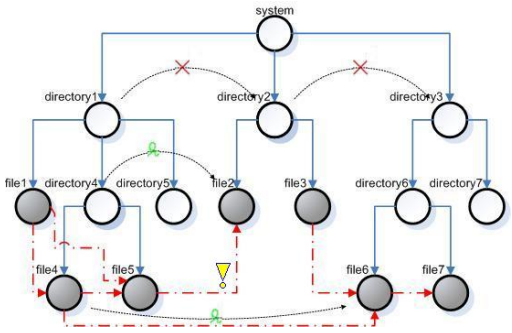
The last two hold for structures that are considered to be *strictly hierarchical* or *layered*. In a strictly hierarchical structure, every vertex of the corresponding graph is coupled to a certain level. As the rules assume, it is in strictly hierarchical structures generally forbidden for include relations to go from a file to a file that is higher in the hierarchy or to go from a file to a file that is more than one level lower in the hierarchy.

6.2 Combining rules

Users are enabled to apply multiple rules on their systems. The set of applied rules can consist of any combination of structural and architectural rules. Consequently, it is possible for rules to be conflicting, which means that the return values of $R(e)$ define different values for a for a specific relation $e \in E_R$. This section explains how this problem is tackled.

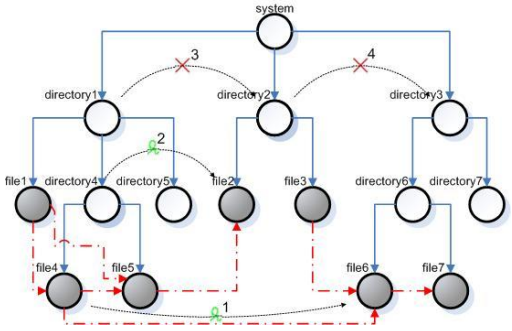
The tool described in this thesis considers architectural rules to dominate structural rules, because architectural rules need to be followed at all times. Consequently, if a relation $e_1 \in E_R$ is in the scope of an allowed structural rule $e_2 \in E_S$ and this relation e_1 violates an architectural rule, then, despite the structural rule, this relation is considered forbidden.

Another problem arises when there are two structural rules that are conflicting. G_{TOT} contains conflicting structural rules $e_1, e_2 \in E_S$, where $e_1 = (v_1, v_2), e_2 = (v_1 v_2)$, with $v_1, v_2, v_3, v_4 \in V_T$ if and only if $((e_1\ is\ allowed \wedge e_2\ is\ forbidden) \vee (e_1\ is\ forbidden \wedge e_2\ is\ allowed)) \wedge ((v_1 \twoheadrightarrow v_3) \vee (v_3 \twoheadrightarrow v_1) \vee (v_1 = v_3)) \wedge ((v_2 \twoheadrightarrow v_4) \vee (v_4 \twoheadrightarrow v_2) \vee (v_2 = v_4)))$. Conflicting rules cause the relations that are in both rules' scopes to be both allowed and forbidden, which, of course, is impossible. The problem is sketched in the following diagram:



This diagram contains two conflicting rules, which causes one relation to be both allowed and forbidden. This relation goes together with an exclamation mark.

The problem can be solved by stating that forbidden rules should dominate allowed rules, which means that the conflicted file relations will always be set to forbidden. Another, more user-friendly approach, is to give the rules an extra attribute that defines the priority of the rules. By manually assigning every rule a unique priority, the effect of the rules is fixed; if two rules are in conflict with each other, then the rule with the highest priority establishes the condition of the conflicted file relations. An example is given in the following diagram:



The numbers next to the rules represent their priorities. In this example, the previously conflicted relation is allowed, because the involved allowed rule has a higher priority than the conflicting forbidden rule.

A list $L = ((c_1, a_1), (c_2, a_2), ..., (c_n, a_n)\ i = 1, ..., n$ is obtained by applying the function R on all rules for a relation $e \in E_R$, where n equals the number of rules. Let this list L be ordered in such a way that each tuple i, with $i = 1, ..., n$, is produced by a rule with a higher priority than a tuple j, with $j = i + 1, ..., n$.

To retrieve the final outcome whether a file relation is allowed or not, the following recursive function is defined:

$$
\begin{array}{lll}
A(i, L) = a_i & & \text{if } c_i \\
\quad\quad true & & \text{if } i = n \wedge \neg c_i \\
\quad\quad A(i + 1, L) & & \text{otherwise}
\end{array}
$$

Stated otherwise, the return value of this recursive function equals the value of a_i of the first rule that has the relation in its scope. If no such rule exists, then the return value equals $true$.

Let $A(L) = A(1, L)$. To know whether a file relation $e \in E_R$ is allowed or forbidden, an ordered list of tuples L should be generated for this relation. Its ordering should be based on the priorities of the rules, taking into account that architectural rules dominate structural rules. $A(L)$ then returns whether the relation e is allowed ($true$) or forbidden ($false$).

Chapter 7

Manipulating system structures

The visualization metaphor makes use of a system's structure to construct the adjacency matrices. There are two different aspects of the tool that require the possibility to manipulate a system's structure and subsequently observe the results in the matrix representations. This chapter explains this feature's relevance.

7.1 Editing structures

The definition of a what-if scenario is a method that is used to forecast values based on information that you provide. The notion of what-if scenarios is based on two different aspects. The first aspect that brought up the need of what-if scenarios is the desire of architects to minimalize the number of violations that exist in their software systems. The number of violations possibly changes as the structure changes, because of the properties of structural rules. Thereby, the violations caused by the architectural rules that deal with the layering of a structure, are also influenced when a strictly hierarchical structure is altered. Potentially, this means that all violations, except for the ones that are caused by the rule that forbids cyclic relations, could disappear if the structure is adequately changed.

Another reason for the use of what-if scenarios concerns component dependencies. Spotting the dependencies between components is easy. One can find all components that include or are being included by a component by checking the corresponding row or column in the matrix representations of these components. When the content of a component changes, its included or including components deserve special attention. By performing what-if scenarios, architects are enabled to follow the movement of these dependencies as the structure changes.

Users must be enabled to edit a system's structure in order to evaluate what-if scenarios. The functions that can be used on the vertices of the structure graph are cut and paste and delete. By using these functions, a structure can be altered, such that the number of violations or dependencies between components decreases.

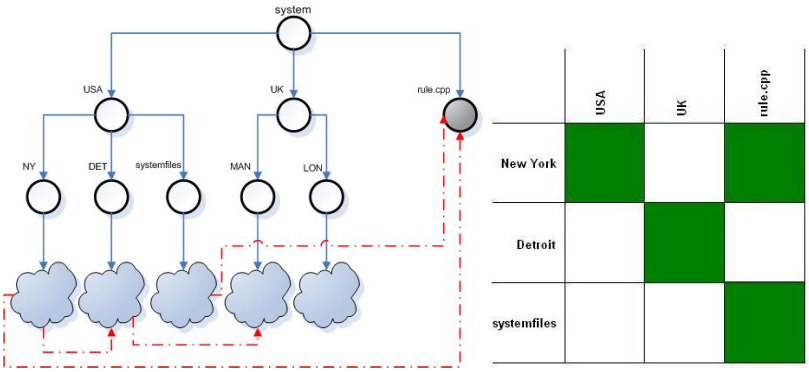
7.2 Defining new structures

Next to the directory structure of a system, companies use several other views or structures on their systems. For architects it is very interesting how file relations manifest themselves in these other structures. The visualization metaphor can work with any software structure, as long as this structure is defined as a rooted tree. As can be concluded from section 4.6, the tool uses a system's directory structure as the initial structure of a system. To enable architects to view the include relations by making use of other system structures, the tool should provide an interface in which the architects can define new structures.

Chapter 4 describes how these structures are represented by rooted trees. It states that some parts of these other structures can be retraced to the directory structure, where other parts are structure-specific. The new structures can be defined as a mutation of the directory structure, supplemented with new vertices. The interface for this part should allow copy and paste functions on the existing structure, together with a function that allows users to introduce a new vertex in the tree.

Next to the fact that the structures should be defined as rooted trees, there are some other demands for these new structures, such that the use of the visualization metaphor makes sense. If a new structure $G_{new} = (V_{new}, E_{new})$ is constructed out of an existing structure $G_T = (V_T, E_T)$, then G_{new} should fulfill the following requirements:

- G_{new} is a rooted tree
- The children of a specific vertex should all be vertices from the existing structure or should all be newly defined vertices. $(\forall v_1 : v_1 \in V_{new} : (\forall v_2, e : v_2 \in V_{new} \wedge e \in E_{new} \wedge e = (v_1, v_2) : v_2 \in V_T) \vee \neg(\exists v_2, e : v_2 \in V_{new} \wedge e \in E_{new} \wedge e = (v_1, v_2) : v_2 \in V_T)$. This claim prevents meaningless matrices, such as presented in the following figure:

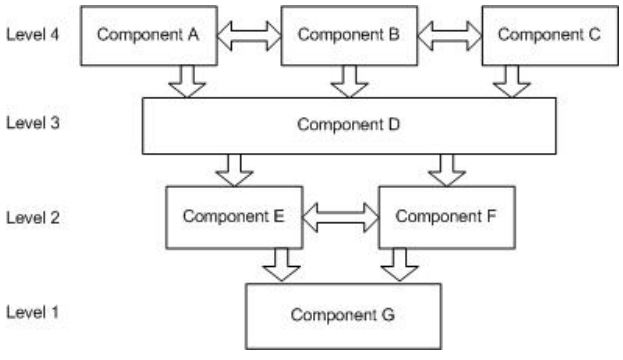


Left is an example of a software system and its file relations visualized in an organizational structure. Right is the corresponding matrix representation. The file *rule.cpp*, the directory *system files* and, possibly, parts of the undefined subtrees come from the existing structure. The figure shows that this system leads to matrices in which apples are compared to oranges.

- If a vertex comes from the existing structure, then so do all its children. $(\forall v_1 : v_1 \in V_{new} \wedge v_1 \in V_T : (\forall v_2, e : v_2 \in V_{new} \wedge e \in E_{new} \wedge e = (v_1, v_2) : v_2 \in V_T))$.

7.2.1 Strictly hierarchical structures

A new structure can be defined as a strictly hierarchical structure, also referred to as a layered structure. Strictly hierarchical structures have their components or building blocks mapped to (integer) levels. All directories and files that are positioned in these components or building blocks inherit these levels. For a strictly hierarchical structure it holds that a file i, with integer lever L_i, may include a file j, with integer level L_j, if and only if $L_i - L_j \leq 1$. An example of a design of a strictly hierarchical software system is given in the following diagram:



A design of a software system that is designed to be strictly hierarchical. This type, where the root's children form the distinct entities in the layering, is implemented in the tool. The levels are presented on the left. The arrows between the components represent the allowed include relations between the components.

In a dialogue with Van Wijk and TIOBE Software, a decision was made to allow users to define levels in a strictly hierarchical structure, though with restrictions: when defining a new strictly hierarchical structure $G_{new} = (V_{new}, E_{new})$, users are only enabled to assign an integer value to the vertices $v \in V_{new}$ that are a child of the root vertex, such that these vertices define the layering of the system. These integer values, which represent the vertices' levels in the layering, do not have to be unique. All descendants of the root's children are consequently located on the same level.

As mentioned in section 5.1.2, there are three different architectural rules. The architectural rule that concerns cyclic relations applies to all system structures. On the other hand are the architectural rules that deal with layering. These only apply to structures that are strictly hierarchical, because these require every file to be assigned to a specific integer level. These rules are evaluated based on the division of levels as defined by the user.

Chapter 8

Other relevant functionalities

The previous chapters deal with the aspects that bring about the most important features of the tool, namely the visualization metaphor, the management of design rules and the possibility to manipulate structures. This chapter handles other relevant features that came up during the course of the project that improve the usage of the tool in different ways.

8.1 Retrieving additional information

Some cells of the matrix representations are colored because of the existence of include relations in the parts of the system that they represent. Uncolored cells represent the absence of such relations. The colorings of the matrices' cells can be used to make assumptions and draw conclusions on the structure and behavior of a system, but it would be nice if also other cell-specific information is visualized. The most important information is the list of all the include relations that instantiate the colored cell. The tool's users should be enabled to distinguish the set of forbidden direct, allowed direct and indirect relations in the list. The first set of relations should be accompanied by the rules that caused the relations to be forbidden, where the last set should come with a textual representation of the path between the two involved vertices. Users should not only be enabled to retrieve this information for every colored cell, but also for the rows and columns of a matrix. The information of a row or column is the combination of the information for all its colored cells.

8.2 Mirroring the matrix

The users that are observing the include relations in their systems are mostly looking at how a component makes use of another component. This can be done in the matrix with the y-axis focussed on one component and the x-axis on the other

component. However, this matrix does not contain any information on whether and where relations are located in the other direction. Depending on the location of the components in the system, it could be quite an undertaking to get to the matrix where the focus vertices have switched by means of zooming. This is another example of how the visualization metaphor does not meet the visualization quality of 'semantic closeness equals visual closeness'. However, to increase the user-friendliness of the metaphor, the tool's users should be enabled to call a function that instantly shows the matrix in which the focus vertices are switched.

8.3 Changing the matrix size

There are two different ways to determine the size of a matrix cell. Firstly, the choice can be made to have a fixed cell size. The size of the matrix then depends on its number of rows and columns; the more rows and columns the matrix has, the larger the matrix will be. The second approach has a fixed matrix size. The size of the célls then depends on the number of rows and columns of the matrix; the more rows and columns the matrix has, the smaller the cells will be.

Both strategies have their disadvantages. The first approach becomes hard to use if the number of rows and columns becomes that large that the matrix does not fit in the user's window anymore. Although nowadays computer screens are large, they have their limits. The second approach becomes hard to use when the number of rows and columns becomes that large that the cells become too small to simply observe them.

The best strategy to determine the size of a matrix cell is considered to be the latter: it makes the zooming actions more comprehensible, because the results are bound to stay within a certain boundary. This is the case for fixed matrix sizes, but not for fixed cell sizes. However, if the cells become to small, this method becomes useless. Therefore, the users should be granted an option that enables users to manually set the size of the matrix and thus influence the size of the cells.

8.4 Show relation quantities

In some cases it would be handy if the cells of the matrix do not only visualize the existence of allowed and forbidden relations, but also the quantity of the relations. For example, if a user uses the tool to make an estimate of the layering of a system, then it is important to know how many relations there are between two components. If there are significantly more include relations from a component A to another component B than there are opposite relations from B to A, then component A is most probably positioned in a higher layer than component B. Consequently, the quantities can also be used to find rules in a system: in the above example there will most probably be a rule that forbids includes from B to A. To enable users to draw such conclusions, it is desired to have an option that enables the visualization of these quantities in the matrix representations.

Chapter 9

Design decisions

The previous chapters are elaborations of the functionalities of the tool-to-be-build. An appropriate system design is made that makes these functionalities feasible. This chapter deals with this design and the major design decisions.

9.1 Server client strategy

Simply said, the tool enables users to retrieve and update information about a system's include relations. Requirement **NFR1** states that the tool should be accessible through the user's web browser. One possible approach is to store all information on the user's computer, such that the user's browser performs all actions and calculations on the data locally. Another, more common approach, is the separation of the graphical user interface part and the calculational part. The graphical user interface is provided in the user's browser, the *client*. The calculational part is performed by another, central computer; the *application server*. Each client can send requests for specific actions or calculations to the application server. The results of these requests are subsequently returned to the client.

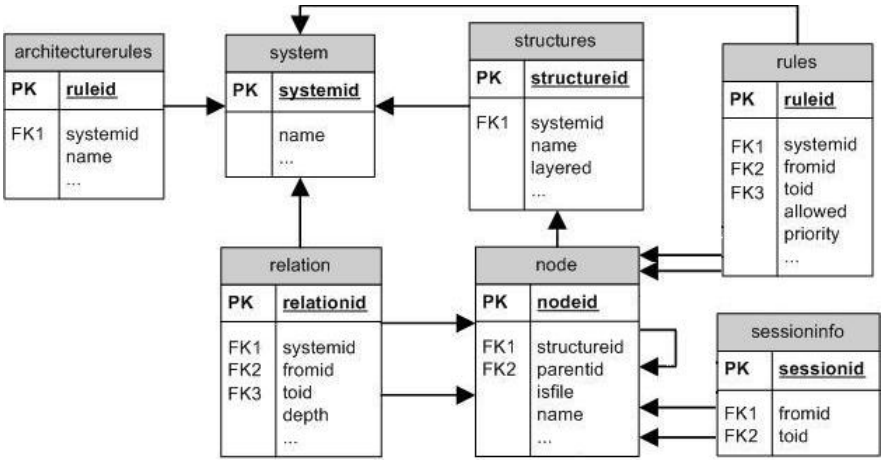
This application server should be able to retrieve and update the required data for every request. Considering the number of users that will work with the same data set from different working stations, it is almost inevitable to have a central data storage that can be consulted for every action or calculation. This data is stored on the so called *data server*. Dependent on the design of a company's internal network, this data server can be the same computer as the application server, but can also be another computer in the network.

9.2 Database design

The use of a database is evident when working in a client-server environment in which the clients work with large amounts of data. The database will be used to store all required data to make all features feasible. The following is a list of all information that is used in the tool and that needs to be stored in the database:

- all introduced systems
- all structures that are introduced for a system
- all direct and indirect file relations that exist in a system
- all rules that are introduced for a system
- the focus vertices of all the tool's users

The first decision to make about the database is about what kind of database to use. The decision was made to use a relational database, because it is the most widely used kind of database in a client-server environment and because the author had quite some experience with relational databases. Next, the database needs a proper design to make the implementation of the tool more easy. As will be explained later in both this and the following chapter, the implemented database scheme slightly deviates from the initial devised database scheme that is depicted below:



The scheme shows the different tables, their dependencies and their attributes. Some of these attributes represent primary keys or foreign keys.

Some explanation on non-obvious items:

- Each *structure* has a boolean attribute *layered*, which equals $true$ if the structure is a strictly hierarchical structure and equals $false$ otherwise
- If a *node* corresponds to a file, then this is stored in the boolean attribute *isfile*

- The *depth* of a *relation* indicates the path length of the relation, which indirectly indicates whether the relation is a direct or indirect relation
- There are two different paths from *relation* to *system* in the diagram. This calls for a database constraint. This constraint claims that he *systemid*'s to which both paths from *relation* to *system* lead, must be the same
- Likewise there are two different paths from *rules* to *system* in the diagram. This requisite constraint implies that the *systemid*'s to which both paths from *rules* to *system* lead, must be the same
- The two attributes *systemid* of the tables *relation* and *rules* are actually redundant, because their values can be retraced via the path to *structure*. This connection defines a structural rule or a relation to be bound to a specific system, such that a simple query suffices to retrieve a system's structural rules or relations.

The feature that enables users to define new structures on a system only came up at a later stage of the project. Unfortunately, there was no time to investigate how rules and relations manifest themselves in different, related structures of a system. Questions that need to be answered are:

- What if one involved node of a file relation does not exist in a structure? The relation can consequently not be defined in that structure.
- What if one involved node of a structural rule does not exist in a structure? The structural rule can consequently not be defined in that structure.
- What happens to a relation in a structure if another structure's rule causes that relation to be forbidden? Should relations be affected by rules that are defined on another structure?

Because of lack of time, a more straightforward solution was devised, such that all features, including the new one, could be implemented. This solution does not define relations and rules to be bound to a specific system, but to a specific structure.

9.3 The exchange format

Users of the tool should be enabled to import systems into and export systems from the database. Users should be enabled to import systems in order to observe the system's include relations with the tool. The export of a specific system can be necessary in several cases. For example, if a system has to be imported in another server or if a specific state of a system needs to be stored. To enable these actions, there must be an appropriate exchange format, such that the information can be extracted from and imported into the database.

As mentioned in section 4.6, the initial data for a system is retrieved from TIOBE Software's quality database which runs on the customers's servers. TIOBE Software uses a query to extract the required information. The output of this query, plain text, is pasted in a file. This file is a list with entries: <*pathname*> / <*filename*> <*TAB*> <*pathname*> / <*filename*>. Every entry in the list represents an include relation from the first file to the second file. By making use of the given absolute paths it is possible to construct the graph that represents the system's structure. Note that in this case, the system's structure is only constructed out of the directories and files that participate in the set of include relations.

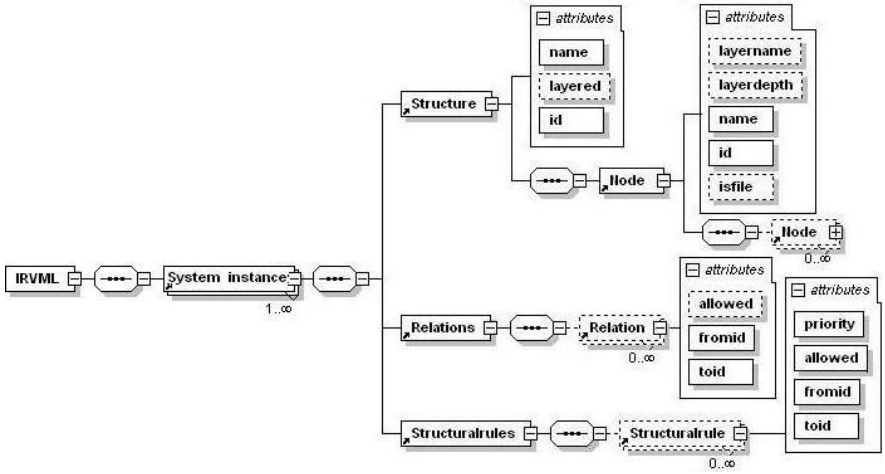
As can be concluded from the previous chapters, a system's structure is defined by three different graphs that represent the structure itself, the relations that exist in the structure and the structural rules that should apply on the structure respectively. Note that the last graph may be empty, because it may be possible that there are no structural rules defined on a structure.

The exchange format should be able to define multiple structures of a system. The input file produced by TIOBE Software does not satisfy this requirement and it does not provide possibilities to define the applicable rules. Because of this, another appropriate exchange format is designed.

Considering the client-server design of the tool, the graphs should be transformed into a format that can be accepted by a browser and subsequently be send to the application server in order to process it into the database. It is common that these graphs are imported by means of a textual description.

Van Ham uses a set of files that contains the required information for his tool. He defined a system's structure in terms of five different abstraction levels. The contents of the abstraction levels are stored in four different files (the abstraction level *System* does only contain the root vertex). The containment/ischild relations between the abstraction levels take another three files. Next to these necessary files, which describe the system's structure, is the file that defines the relations between two *Classes*. Van Ham also makes a distinction between allowed relations and forbidden relations. The forbidden relations between the different abstraction levels are stored in a distinct file. Unfortunately, this all takes a dozen of files, which amounts to a complex whole.

Of course, it is possible to paste the contents of all these files in one file, but this would lead to one big, complex file. A better approach would be one that establishes a well-organized file that makes sense to its readers. The use of an XML jacket is a good solution, because of XML's simplicity (it is readable and understandable), extensibility (the format can easily be extended), interoperability (XML is supported by all browsers and platforms), openness (XML can be displayed with many programs) and because a definition (an XSD) can be made for every XML format. The following diagram represents the XML definition that is designed such that the XML jacket can embody all required information:



A visual representation of the XSD.

Every XML format has its own name, like HTML and SVG. The XML that is used by the tool is referred to as IRVML, which stands for Include Relation Viewer Mark-up Language. The XML format uses an *IRVML* element to define a system by several *System instances*. A *System instance* is defined by a *Structure*, a set of *Relations* and a set of *Structural rules*. The attributes that belong to these entities are also included in the figure. The entities that have a dashed border are optional.

By making use of XML parsers, the application server can parse the contained information into the database quite easy. Nowadays, must programming environments come with libraries that support XML parsing.

Because it is possible that multiple structures are defined on a system, there should be two options for a user. First, a user should be enabled to retrieve the IRVML for the system's structure that he is currently viewing. Second, the user should be enabled to extract the IRVML of all a system's structures.

9.4 View and editor mode

The functional requirements in chapter 3 convey the two distinct use cases that hold for the tool: in most cases, the users will simply want to observe the include relations of a system. In addition to this, there are the users that want to change a system's data by defining rules on a structure and by altering a structure. In other words, there are users that want to view and users that want to edit.

The tool should provide two different modes, such that users that just want to browse their systems are not flooded by the extra options that are used by editing users. These two modes are defined as *view mode* and *editor mode*. Upon selecting a specific system and structure to visualize the include relations, the user should also choose whether he wants to go into view or editor mode. Users should be enabled to switch their mode while using the tool.

9.4.1 View mode

Users that just want to use the tool to observe the include relations of a system should be enabled to do the activities listed below:

- **[VM1]** Use the visualization metaphor to observe the include relations
- **[VM2]** Retrieve additional information of specific part of a matrix
- **[VM3]** Find out where his focus vertices are in the current structure
- **[VM4]** Manually set his focus vertices
- **[VM5]** View the rules that apply on a system
- **[VM6]** Toggle between different structures of the current system
- **[VM7]** Enable the visualization of the transitive relations
- **[VM8]** Enable the visualization of the relation quantities
- **[VM9]** Change the size of the matrix representations
- **[VM10]** Mirror the current matrix representation
- **[VM11]** Output the IRVML of the structure currently viewed
- **[VM12]** Output the IRVML of the whole system currently viewed
- **[VM13]** Go into editor mode

9.4.2 Editor mode

Users that want to manipulate a system by changing its set of applied rules or by editing one of its structures must be enabled to do the following:

- **[EM1]** Everything that is possible in view mode, except for the last activity.
- **[EM2]** Edit the current structure
- **[EM3]** Add a new structure for the current system
- **[EM4]** Add and remove a structural rule for the current system
- **[EM5]** Redefine the priorities of the structural rules of the current system
- **[EM6]** Filter out the effect of an architectural rule of the current system
- **[EM7]** Go into view mode

9.5 System instances and user sessions

After the XML input is imported in the tool and the necessary processing has been performed, the new system can be used in the tool. This section handles the notions of system instances and user sessions, which play an important role in the use of the tool.

9.5.1 System instances

Functional requirement **FR6** indicates the reason for the manipulations of a system's structure. It is plausible to assume that most manipulations on a system's data are attempts to decrease the number of violations or component dependencies. It is highly unwanted if the observations of users are influenced by these attempts, because these users want to view the current state of their software systems and do not want to view altered systems.

As can be concluded from the previous section, there is a difference in functionality between view and editor mode. The users in editor mode are enabled to manipulate a system's data. Considering the size of the data and the required calculations that need to be done, it is possible that these manipulations take considerable time, ranging from minutes to days. A problem arises when other users are working with the same dataset at the same time as that the server is performing these calculations on the dataset: the correctness of the data cannot be guaranteed.

The above two problems indicate the need for a separation of datasets; the users that are performing attempts to improve a system should work with a different dataset than the users that are simply browsing the dataset.

As will be explained in section 12.1, an important possible feature of the tool concerns version management. The feature makes it possible that different versions of a specific system coexist in the tool. As a result of this, a user of the tool should not only choose a system, but also a specific version of that system, before he is enabled to observe the include relations of that specific system. This aspect brings about the notion of system instances. Such a system instance represents a specific version of a system and should be selected to observe that version's include relations. Each system instance makes use of its own dataset.

Hereby, this concept can be used to solve the problems stated above: if a user wants to perform improvement attempts on a specific system instance, then he should create a copy of that instance, which becomes another instance of that system. Consequently, the user can use the new instance to perform the improvement attempts without causing any problems to other users.

9.5.2 User sessions

Users of the tool have their own so-called session. This session represents the information that defines his state. Some of this information is stored on the server, where other parts are stored in the user's browser, the client. A user session is started every time that a user uses the tool.

The following is a list of the information of a user session:

- The system instance that the user is currently using
- The structure of the system that the user is currently using
- The two focus vertices that represent which matrix the user is viewing
- Whether the user is in editor- or view mode
- Whether transitive relations should be visualized
- Whether the relation quantities should be visualized
- The size of the matrix representations

The first three items describe the information that will be stored on the server side, where the other information will be stored in the user's browser, the client. As can be concluded, the information on the server defines what needs to be visualized and the information on the client defines how it should be visualized. This is done because the information on the server is required by the actions and calculations that are being performed on the server and the information on the client is required by the actions and calculations that are being performed on the client.

9.6 Separation of concerns

The functionality of the tool can be split up into six different parts. The following list contains the names that are used to refer to these parts. Furthermore, the list specifies these parts' functionalities in terms of the activities as listed in section 9.4.

- The **include relation viewer** part
  - **VM1**: The matrix representations should be contained in this part
  - **VM2**: To retrieve the additional information of a specific part of a matrix, the part should be selected in the matrix
  - **VM4**: The focus vertices should be changed when zooming actions on a part of a matrix
  - **EM4**: By selecting a specific part of a matrix, the user should be enabled to define a rule on the corresponding vertices

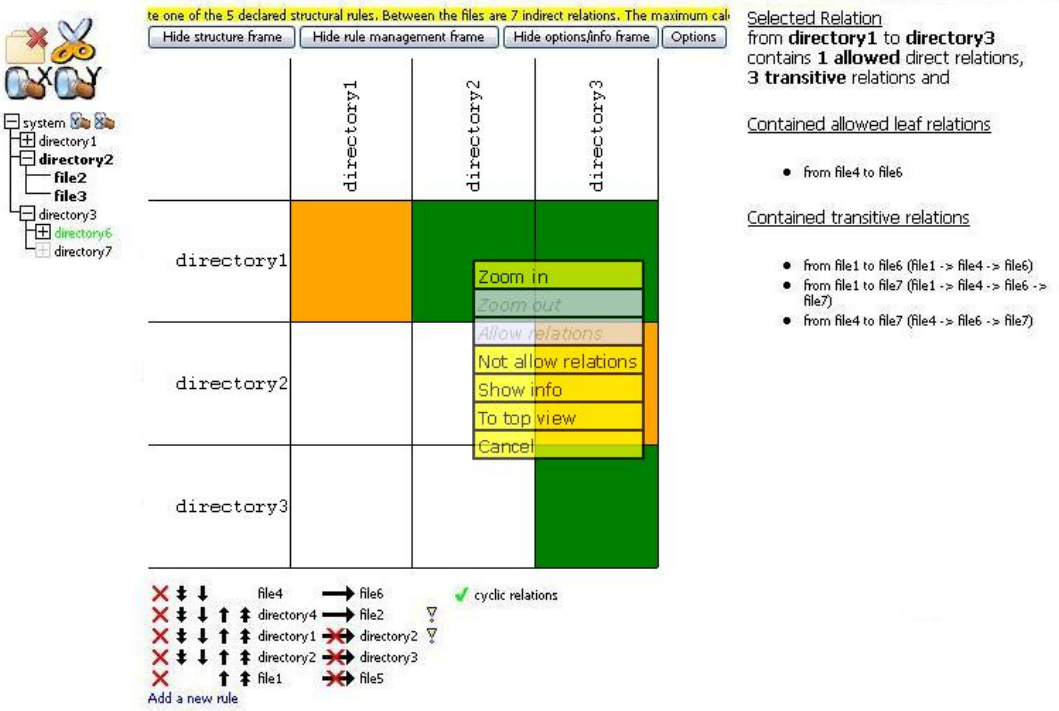
- The **additional information** part
  - **VM2**: The additional information of a specific part of a matrix should be visualized in this part
- The **rule management** part.
  - **VM5**: This part should visualize the applied rules
  - **EM4**: Users should be enabled to add and remove structural rules by means of forms
  - **EM5**: Users should be enabled to change the priorities that belong to the applied structural rules by means of forms
  - **EM6**: Users should be enabled to toggle between whether an architectural rule should be applied
- The **user options** part
  - **VM6**: The visualization metaphor's used structure should be changed when selecting another structure of the current system
  - **VM7**: Users should be enabled to toggle the visualization of transitive relations on or off
  - **VM8**: Users should be enabled to toggle the visualization of relation quantities on or off
  - **VM9**: The user should be enabled to give the new size for the matrix representations, such that the matrix becomes larger or smaller
  - **VM10**: The focus vertices of a user should be changed when performing this action
  - **VM11**: This part should generate the IRVML of the current structure when the corresponding function is called
  - **VM12**: This part should generate the IRVML of the whole current system when the corresponding function is called
  - **VM13**: By changing modes, the tool's functionality should be changed
- The **structure management scenarios** part
  - **VM3**: The current structure should be represented by means of a tree-view. The user's focus vertices should be indicated in this tree-view
  - **VM4**: A user should be enabled to change one of his focus vertices to another vertex by using the tree-view of the current structure
  - **EM2**: This part should enable users to edit the current structure by allowing cut and paste and delete functions on vertices of the structure
- The **new structure introduction** part
  - **EM3**: This part should show the original and the new structure by means of a tree-view. The new structure should be created by adding and removing vertices and by copying vertices from the original structure and pasting them into the new structure

These different parts enclose the whole functionality of the tool. The user-friendliness of these parts strongly depends on how these functionalities are offered to the users. Because of the interdependencies between the different parts, the functionalities should be as visually close together as possible.

Chapter 10

The graphical user interface

The following figure shows how the different parts are combined in the user's window. Each part uses its own HTML frame.



A screenshot of the tool. All different parts take their own frame.

The screenshot shows that the user's window contains four of the six existing parts. The leftmost part of the window is used by the structure management part. The

part at the bottom of the window represents the rule management part. The rightmost part is filled with the additional information of a part of the matrix that is visualized in the center of the window, in the include relation viewer part. The new structure introduction part takes its own window, while the other missing part, the user options part, makes use of the same frame as the additional information part. The following subsections are elaborations of the different parts.

10.1 The include relation viewer part

The include relation viewer part embodies the visualization metaphor. A zoom-in (zoom-out) action can be initiated by a short single (double) click on a specific part of a matrix. The metaphor is extended with an option list that can be invoked for each part of a matrix. This list can be invoked by holding the mouse button down for over a second on a specific part of a matrix. An example of such an option list is represented in the following figure. If the user is in view mode, then the option list is adjusted to this situation by leaving out the options that are only allowed in editor mode.

The option list that belongs to a specific part of a matrix. Two of the options are disabled because they are not possible for the selected part of the matrix. The shown option list is invoked by a user in editor mode. If a user is in view mode, then the "Allow relations" and "Not allow relations" options are absent.

The option list shows the option to go to the top view, which sets the user's focus nodes to the current structure's root node. This feature can be used if a user becomes lost in the maze of matrices or if he just wants to go back to the top matrix. Users are enabled to view the additional information of the selected part of the matrix by clicking on the "Show info" option. This information is shown in the part described in the following section.

The include relation viewer part also enables users to define rules on colored cells or on rows or columns that contain colored cells. If the colored cell (multiple cells if rows or columns are involved) is green, then the user can define a forbidden rule on the corresponding nodes, which causes all involved relations to become forbidden. If it is red, then the users can define an allowed rule, whereas he can choose out of both options if the cell is colored orange (or, in the case of rows and columns, if it contains both red and green cells or an orange cell). The new rule is added in the rule management part and is consequently assigned the highest priority. The results become visible in the matrix, when the needed calculations have been performed.

10.2 The additional information part

If a user requests additional information of a specific part of a matrix, then this information is shown in this part. The additional information contains the following:

- The number of enclosed allowed direct, transitive and forbidden direct relations
- A list of all forbidden relations, of which each relation is followed by the set rules that caused the violation
- A list of all allowed relations
- A list of all transitive relations, of which each relation is followed by the path that connects the two involved nodes

10.3 The rule management part

The rule management part is separated in two distinct lists, namely the list that represents the structural rules (on the left) and the list of architectural rules (on the right). This way, the difference between the two types of rules is made clear. The list of structural rules is ordered, based on the priorities of the rules. The rule on the top of the list has the highest priority, where the rule on the bottom has the lowest priority.

The list on the left contains red crosses, a number of vertical arrows, two names that are separated by an horizontal arrow and some exclamation marks. The red crosses can be pressed to remove the corresponding rule, where the vertical arrows can be used to manipulate the priorities of the structural rules. The two names define the two nodes on which the rule is based. The horizontal arrow has a red cross through it if the rule is forbidden, where it has no cross through it otherwise. The exclamation mark represents whether the rule is in conflict with another rule. Only users in editor mode are enabled to change the applied structural rules.

The list of architectural rules consists of two different parts, namely the description of the rule and a check mark that shows whether the rule should be applied or not. If not, then the check mark is crossed out, where it is not otherwise. Only users in editor mode are enabled to change this by pressing the check mark.

The icons used in this part give extra information when the user holds his mouse pointer over them. For instance, the horizontal arrows in the list of structural rules give more information about the rule and the exclamation marks give an explanation of the conflicts.

The link below the list of structural rules is available for users in editor mode. This link gives access to the form that can be used to define a new structural rule. The form is displayed in the following figure:

The form that enables users to define a new structural rule.

The two drop-down boxes can be used to set the involved nodes. The boxes are filled with the nodes that form the rows and columns of the current matrix. The down and up buttons can be used to change the corresponding focus node into the selected node, respectively its parent. The radiobuttons should be used to define whether the rule is an allowed or a forbidden rule. The textfield can be used to give a description for the rule. By pressing the button "Add new rule" the rule is added and the necessary calculations are performed.

The effects of the changes that are carried through by a user in editor mode are visualized in the include relation viewer part when the required calculations have been performed.

10.4 The user options part

This part of the tool can be used to change the user options of the tool. This part uses the same frame as the additional information part and can be invoked by pressing the "Options" button at the top op the window. The different options are sketched in section 9.4 by the activities **VM6** to **VM13** and **EM9**, where **VM13** and **EM9** are the opposites of each other. The lay-out of this part is sketched in the following figure:

The lay-out of the user options part.

The lay-out of the user options part is quite self-evident. The boolean user options are represented by checkboxes, where the other activities are initiated by pressing the corresponding links. Most of these links invoke forms that require input from the user in order to perform the indicated action. The link "I want to create a new structure" invokes the window that contains the new structure introduction part after the user has given the proper input.

10.5 The structure management part

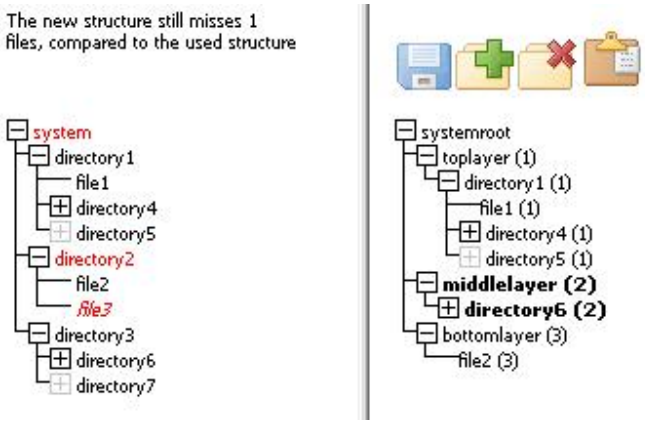
The structure management part can be used for several purposes. Its main purpose is to edit a system's structure. The currently viewed structure is represented in a tree-view. The nodes in this tree-view can be selected. The icons at the top of the tree-view represent the actions that the user can perform on the selected node. The user can choose to cut, paste and delete nodes of the current structure.

The calculations that are the consequence of a cut and paste action are divided into two parts. The first part calculates the resulting structure and subsequently shows the results by means of the tree-view, where the second part calculates the effects concerning the structure's rules and the nodes' Depth First Search values. The calculations performed by the second part are initiated once the "save" icon is pressed. This enables users to perform multiple cut and paste actions before the more time consuming calculations are performed. The results will be visualized in the include relation viewer part when all necessary calculations have been performed.

The tree-view can not only be used for editing, but is also used to indicate the user's focus nodes, which are accompanied by small magnifying glasses with the corresponding axis in it. If a node has been selected, then the two magnifying glasses at the top of the tree-view can be pressed to manually set the corresponding focus node to the selected node.

10.6 The new structure introduction part

This part enables users to define a new structure for the currently viewed system. To define a new structure, the user has to press the corresponding link in the user options part. Furthermore, the user has to give a name for the structure and specify whether the structure will be a strictly hierarchical structure. Before defining a new structure, the user has to select an already introduced structure on which base the new structure will be constructed. The following figure shows the lay-out of this part:



The lay-out wherein a new structure can be introduced.

The lay-out is divided into two parts; a part that contains a tree-view of the original structure and a part that contains a tree-view of the new structure. The possible functions, copy, paste, add, delete and complete, are represented by the icons above the tree-views. The functions can only be performed if the new structure will satisfy the rules that hold for system structures. If all files that are descendants of a node of the original structure are located in the new structure, then the node is colored black, where the node is colored red otherwise. This information can be used to conclude whether the new structure is finished or not. If the new structure is a strictly hierarchical structure, then its nodes are accompanied with their assigned layer levels.

A new structure becomes available in both view and editor mode when the necessary calculations have been performed. Note that these calculations are almost similar to the calculations that are performed when a new system is introduced to the tool.

Chapter 11

Implementation

This chapter is an elaboration of the implementation according to the design and its decisions, which are explained in the previous chapter. The chapter begins with a section that handles some properties of the code. The following two sections deal with the problems that were encountered during the implementation. These problems mostly relate to the non-functional requirements stated by TIOBE Software. The solutions to these problems lead to a new database design, which is also discussed in this chapter. Furthermore, the chapter handles two graph operations that deserve special attention.

11.1 Code properties

The calculations and actions performed on the server side are realized by PHP scripts. The 79 scripts contain over a $4,000$ lines of code. The used type of relational database is MySQL. The choice was made to use a combination of PHP and MySQL, because TIOBE Software's framework runs in the same environment. By developing the tool in this environment, the integration of the tool in the framework would take less effort.

As mentioned in subsection 9.5.2, there is some data that needs to be stored on the client's side. This data is stored by making use of Javascript variables. Besides, Javascript is used to control the dependencies between the different parts, i.e., the different frames. The include relation viewer part includes a .js file that declares these variables and functions that are on the client side. The other frames can also make use of these functions by means of cross-frame scripting. The included .js file contains 275 lines of code. Furthermore, the client side makes use of a .CSS file, that defines the stylesheet used by the tool.

11.2 Making the visualization metaphor web-enabled

The main part of the tool uses the visualization metaphor of matrix representations to visualize the include relations of software systems. Matrixzoom already encloses this metaphor nicely, but it is a stand-alone application. TIOBE Software's hard requirement is to make the tool web enabled (**NFR1**) and they want that e-verybody within the same network should be enabled to use the tool without having to install other software (**NFR2**). Next to these requirements is the performance issue (**NFR3**). This section handles the attempts that were done to fulfil all these non-functional requirements.

11.2.1 Failed strategies

Before the right strategy was found, three other strategies were examined and partly implemented. All strategies are web-based approaches of the visualization metaphor, but they did not meet the imposed performance requirement.

HTML DIV elements

The first approach was to implement the visualization metaphor by just using existing HTML elements. Matrices can be considered to be constructed out of cells. This strategy uses a <DIV> element for every cell in the matrix. A <DIV> element is an element that is used to define a distinct block (or division) in a HTML page.

These <DIV> elements are standard in HTML and are acknowledged by all important browsers. Their properties make these <DIV> elements interesting. Every <DIV> element can be assigned its own absolute position in the window, along with its own width and height. Thanks to these properties, this approach is able to deal with the zooming part of the visualization metaphor. Also, a <DIV> element can have its own color. So, the <DIV> element can get colored, if the corresponding relation exists.

However, when viewing a fifty by fifty matrix, then this approach requires 2500 <DIV> elements. Next to the fact that this requires much data transfer, zooming actions give problems. When zooming in on a cell, it requires that all other cells shove aside and grow larger. So, the animation requires twenty five hundred HTML elements to be altered (and now the underlying matrix is not even in discussion). Unfortunately this is way too much asked from present computers.

Nevertheless, hopes were not already lost. Instead of approaching matrices as a composition of cells, one could say that they are constructed out of lines. Also this approach was implemented. A <DIV> element is used for each line and each colored cell. Now, taking again a fifty by fifty matrix with fifty colored cells, the browser should calculate a zoom in effect for 152 <DIV> elements. Unfortunately, this also is too much asked from present computers.

While experimenting with this last approach, it came to notice that the problem of this approach is the number of colored cells that is shown in the zoom in effect. The approach works a lot better when using less colored cells. This is why a third attempt was done for this approach. When a cell is selected for zooming in, a part of the context disappears, such that the number of colored cells is rather low. This approach works quite well, but loosing context makes the metaphor less powerful.

HTML TABLE elements

The next step attempt was to use <TABLE> elements. These elements are also standard in HTML and are used to construct tables in a HTML page. The idea is to use a table element for each matrix. So, when doing a zoom action on a cell, the HTML document contains two tables.

The advantage of using TABLE elements for the rendering of the matrix abstractions is the straightforward relation between matrices and tables: matrices are tables. A table in HTML is constructed out of <TR> and <TD> elements, which resemble the rows, respectively the columns of the table. Each cell has its own properties; for example, a color. Another advantage is the easy way to establish the zoom in effect on a table. By simply changing the table's width and height, all its cells grow proportionally.

Although it seems a trustworthy approach, the performance is still a problem. For a twenty by twenty table, the frame rate is already very low, which means that a fluent zoom in effect will never be obtained using the <TABLE> elements of HTML.

GIF generation

After these two approaches, the solution was sought in another direction than built-in HTML elements. The next approach concerned the generation of GIF files.

GIF files are commonly used in HTML and are acknowledged by all standard browsers. GIF files can easily be created, because GIF creation and editing is encapsulated in most programming environments. GIF was designed as an efficient means to transmit images across data networks; the GIF is generated on a server and send to a client. The client side is only responsible for visualizing the matrices, while the server side is responsible for the storage of the dataset and, in this approach, the creation of the GIF files.

The idea arose to precalculate all GIF files. This meant that all possible matrix abstractions are calculated in advance. When a client requests a specific matrix abstraction, the server searches, retrieves and sends the corresponding GIF file. Leaving out the possibly large amount of required disk space, this would work if the data is static. However, looking at the features of the rule management part and structure management part, this is not the case. Dynamic data requires the GIF files to be generated on the fly. In this approach PHP is used on the server side to create the GIF files. The time between a GIF request and the display of the

GIF is approximately 0.25s, which is very reasonable.

However, this approach does not work when a zoom in effect is created. A zoom in effect is created on the client side by simply increasing the width and height of the GIF. But, along with everything inside the GIF, also the lines in the GIF grow proportionally. If a zoom in effect of fifty times is required, a line with a width of one pixel grows to a line with a width of fifty pixels, which is undesirable.

Another approach was to generate the complete zoom in effect with GIF files. For a fluent zoom in effect, approximately fifteen GIF files are required in one second. Although it was known that the generation of one GIF required 0.25s, it was still given a shot. Unfortunately, after a request for a zoom in effect, it takes about three seconds before the animation is shown, because the server needs this time to make all required GIF files.

11.2.2 Successful strategy

An upcoming technique for displaying images using browsers is SVG. Although SVG is not yet supported by all common browsers, it is used more and more in websites. SVG is not only suited for static images, but offers also possibilities for animation.

What is SVG

SVG stands for Scalable Vector Graphics. SVG is a language, defined by XML, which mainly is used for the representation of two-dimensional graphics. Specifically, SVG is used to create vector graphics for display on computer systems. An SVG graphic consists of geometric forms, such as points, lines, circles, polygons and curves, text and embedded digital images. These graphical elements can be displayed, transformed, animated, textured, shaded and made interactive. Because SVG is an XML variant, it is possible to view and edit SVG files with a simple text editor. Next to this XML-based file format, SVG comes with a programming API for graphical applications.

SVG is an open standard, which means that its specification is publicly available. This specification is controlled by W3C and can be found at $http://www.w3.org/TR/SVG/$. Thanks to this open standard, many SVG freeware and commercial software tools are available from a wide variety of developers and vendors. Another important feature of SVG is its interoperability.

Future of SVG

As a standard language for vector graphics, SVG has great potential. Thanks to the fact that SVG is an open W3C standard, SVG will enjoy widespread support as one of the main pillars of Web graphics. The Flash technology, which offers similar

functionality as SVG, is not an open standard, which seemed a strong holdback. For SVG, it is most probable that many different developers will support SVG, since it won't be under the control of a single company.

The major drawback of SVG at the moment is that no browser fully supports SVG. This means that SVG files cannot be viewed in standard browsers. An installation of an external piece of software is required to enables browsers to display SVG documents. However, the future of SVG seems bright: Mozilla and Microsoft plan to fully support SVG.

Change in hard requirements

Using the knowledge that was gained during the tests with GIF generation, simple tests were done with on the fly creations of SVG animations. The test results were satisfactory, because nice zoom effects were created on predefined matrices. Knowing this power of SVG animation, the tests were taken to a next step. A test dataset was created that described a physical architecture of a very small software system, along with some include relations. This test set was stored in a MySQL database on a server and a client was used to browse this information via the visualization metaphor. Using SVG, nice zoom in and zoom out effects were obtained. With these satisfactory results, the decision was made to continue the project using SVG for the visualization of the include relations.

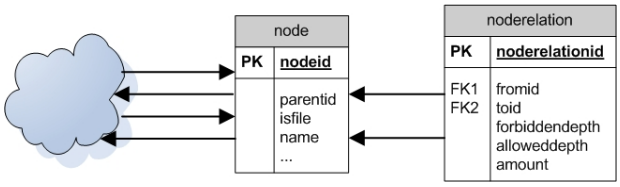
As stated in the previous chapter, TIOBE Software wants the tool to be available through the web without having to install other software **NFR2**. However, because SVG is not fully supported by any browser, users are forced to install software for viewing of and interaction with the SVG. This piece of software is an external SVG viewer that integrates with the used browser as a plug-in. Unfortunately, this is a necessary step for users to take, until their browsers fully support SVG.

11.3 Performance

The most important requirement deals with the performance of the tool-to-build. The tool embodies some features that become significantly slower as the used dataset grows larger. This problem not only concerns the include relation viewer part, but also the rule management part and the structure management part. The problem still existed after these part's algorithms were speeded up. The solution had to be found in the usage of the database. This section handles the adjustments that were made to the database and consequently in the used algorithms, such that the performance became acceptable. The resulting database is described in the following section.

11.3.1 Precalculation data for view mode

The biggest problem concerned the speed of the SVG creation on the application server. To determine which cells of a matrix should be colored and how they should be colored takes some intensive calculations on the dataset. Most users will work in view mode, because the most prominent use case concerns the browsing of the data. To reduce the required calculation time, the tool precalculates the necessary data for every structure. The table *noderelation* is used to store this data. Its structure is depicted in the following figure:



The figure represents the structure of the table *noderelation* that contains the precalculated data that is necessary to create the SVG files in view mode.

The table stores all node tuples that have at least one include relation within their corresponding system abstraction, so it is not exhaustive. This way, the table can be consulted for every cell of all possible matrices whether and how it should be colored. The *fromid* represents the corresponding node on the y-axis, where the *toid* represents the corresponding node on the x-axis. The attribute *amount* stores the number of relations that exist within the corresponding system abstraction. The cell should be colored if *amount* is greater than zero. The attributes *forbiddendepth* and *alloweddepth* store the minimal depth of the involved forbidden relations, respectively allowed relations. Because transitive relations can not be forbidden, the value of *forbiddendepth* equals either zero or one. How the cell should be colored depends on the values of these two attributes.

A consequence of the precalculation of the needed data is that the data does not match the actual state of the structure if some other user has changed the data of the structure. To guarantee that users in view mode are not working with invalid data, the view mode is disabled when a structure is changed. The view mode can only be enabled if the data is recalculated again.

A distinction is made between the precalculation of the data of direct relations and that of transitive relations, because not everyone is interested in the transitive closure of a system's include relations and because the precalculation of the data takes quite some time, especially for the transitive relations. Consequently, the visualization of transitive relations is unavailable in view mode until the required data is precalculated.

11.3.2 Separate direct and indirect relations

It is easy to imagine that the number of indirect relations is much larger than the number of direct relations that exist in a system. The number of indirect relations strongly depends on the structure of the systems, but experience learned that the ratio between direct and indirect relations is about one to ten. The speed of the SVG creation is significantly lower when the transitive relations are added to the set of direct relations, because the efficiency of the queries strongly depends on the number of relations in the table *relation*. Because not everyone is interested in the transitive relations, the set of relations is split into two different tables, named *leafrelation* and *transitiverelation*, by which the performance of the queries increases. Users that do not want to view the transitive relations only work in the table *leafrelation*, where they work in both tables if they do want to view the transitive closure of their system.

11.3.3 Depth first search

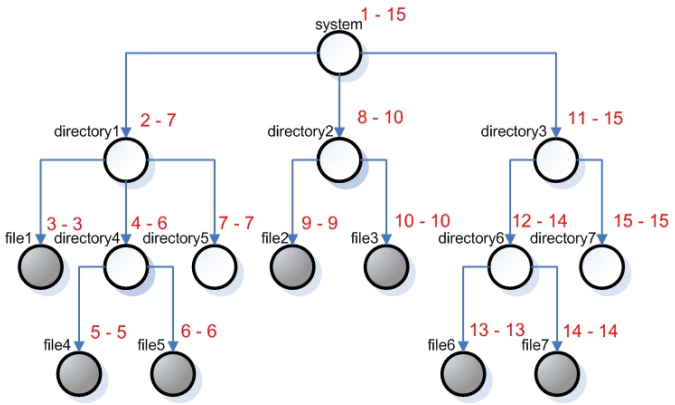
As mentioned, the task to find out whether and how a cell is colored, is time-consuming. The task implies to find all the relations that exist between the subtrees of the corresponding nodes. With the precalculation of the data, this task is simplified in view mode, but it is still a time-consuming task during the creation of SVG files in editor mode and during the precalculation of the data for view mode.

The implemented approach requires two extra attributes for each node of the structure tree. The values are attributed by an algorithm that runs through the structure tree by means of a Depth First Search, starting at the root node. The first value that is assigned to a node, *DFSvalue*, corresponds to its sequence number of the in-order tree walk. The other value, *DFSrangeto*, corresponds to the highest attributed *DFSvalues* of the node's descendants. Below is the pseudo-code of this part of the algorithm:

dfs(v, i)
1. $i = i + 1$
2. $v_{DFSvalue} = i$
3. for each vertex v_1 in $Adj[v]$
4. $\quad i$ = dfs(v_1, i)
5. $v_{DFSrangeto} = i$
6. return i

Here, $Adj[v_1]$ is the set $\{v2 \in V_T | (\exists e : e \in E_T : e = (v_1, v_2))\}$.

The first run of this recursive algorithm is induced by dfs(v, 0), with $v \in V_T$ as the root of the tree. The following figure shows the result of the algorithm:



A small software system depicted as a node-link diagram. The two attributed values stand next to the nodes.

The algorithm that retrieves the include relations within a specific part of a system makes use of these two values. The include relations that exist within the system abstraction defined by the nodes v and u, with $v, u \in V_T$, $v_{DFSvalue} = a$, $v_{DFSRangeTo} = b$, $u_{DFSvalue} = c$ and $u_{DFSRangeTo} = d$, are those relations $e \in E_R$, with $e = (i, j)$, where $i, j \in V_T$ and $a \leq i_{DFS} \leq b \wedge c \leq j_{DFS} \leq d$. This means that the complexity of the algorithm equals $O(n_{relations})$, when a specific cell's coloring is investigated. The complexity of this algorithm equals $O(n_{nodes}^2 * n_{relations})$, when all combinations of nodes are examined.

The advantages of the algorithm lies within the efficiency of the database: the filling of the table *noderelationsi* is done by a small number of different queries. These queries have a high performance due to indexing on the involved attributes. "An index is a feature in a database that allows quick access to the rows in a table. The index is created using one or more columns of the table. Not only is the index smaller than the original table (due to having fewer columns), but it is optimized for quick searching, usually via a balanced tree" **[11]**. The used indexes cause the implemented algorithms to have a higher performance than the previous complexities do assume.

Another possible algorithm loops through all file relations. A temporary $n_{nodes} \times n_{nodes}$ table, located in the server's memory, is used to store the information of the file relations that exist in the cells' corresponding system abstractions. The algorithm changes the concerned cells of the temporary table for each file relation. If the average depth of the structure tree is considered $log(n_{nodes})$, then the complexity of this algorithm would be $O(log(n_{nodes})^2 * n_{relations})$, which assumes that this algorithm has a less complexity than the algorithm described above. This algorithm was also implemented, such that the results of the temporary table are inserted in the table *noderelationsi*. For three different systems, both algorithms were used to precalculate the data for view mode, leaving out the data for transitive relations. The algorithm that uses the assigned DFS-values takes about 10-15 minutes, where the algorithm that loops through all file relations takes more than a couple of hours. When the data for transitive relations is also included, the first algorithm takes 20-30 minutes, where the other algorithm takes more than one and a half day. This was reason enough to dismiss the last algorithm.

11.3.4 A distinct set of tables for every structure

Until now, all information of nodes and relations that are part of a system are put into four tables that are shared by all structures. These are *node*, *leafrelation*, *transitiverelation* and *noderelation*. As the number of introduced systems increases, the number of entries of these tables grows larger and larger. The required calculations, i.e., queries, are decelerated enormously due to these large tables.

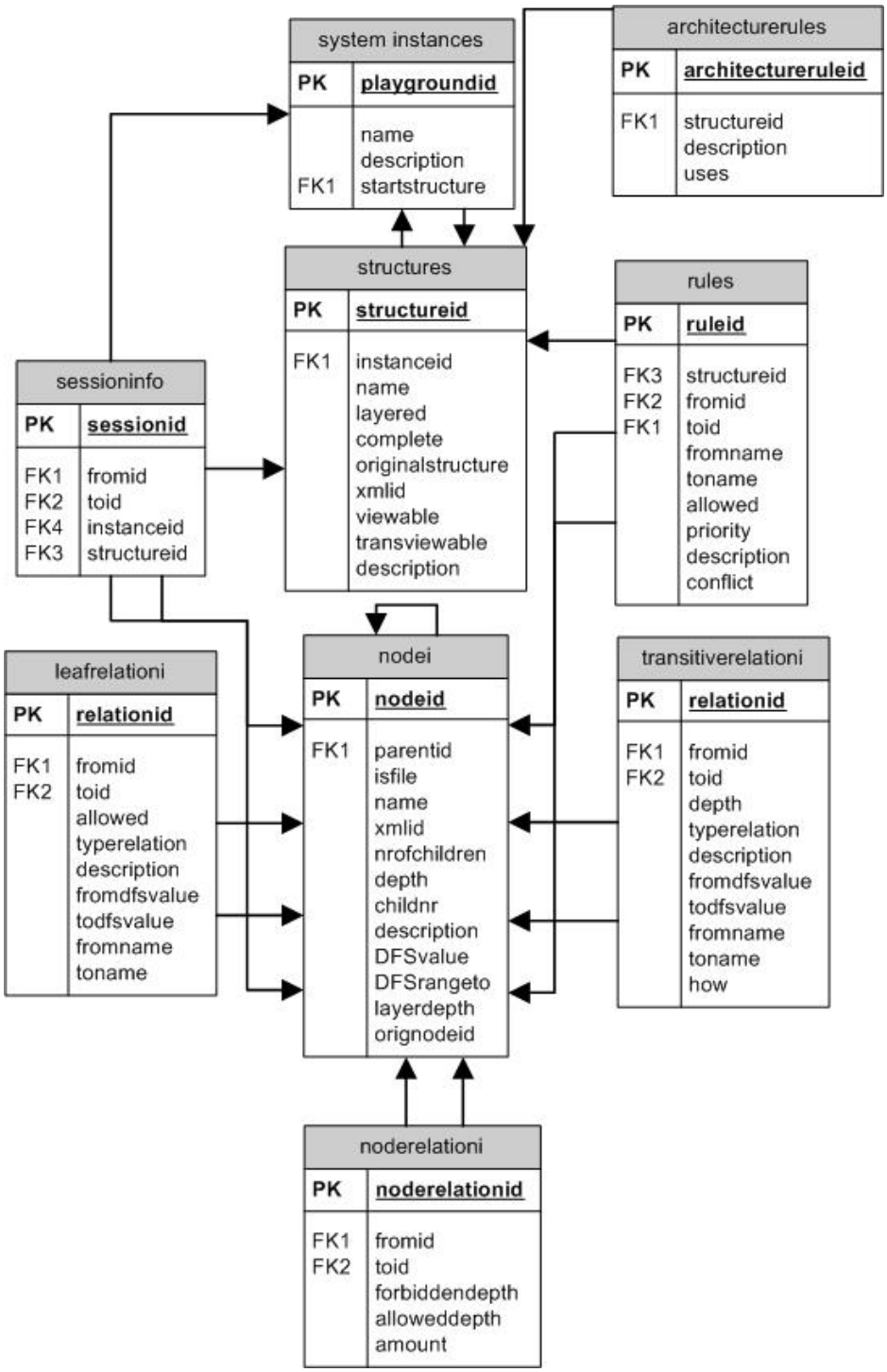
Therefore, a distinct set of tables is defined for each introduced structure. Every structure gets its own *node*, *leafrelation*, *transitiverelation* and *noderelation* table, such that the number of entries in these tables stays relatively low. The table names are followed by an integer value that resembles the *structureid* of the structure they belong to.

11.3.5 Redundancy in database

For large systems another problem arises that also concerns performance. Most calculations that concern include relations or structural rules, require a query that takes the product of several tables; for example, queries that retrieve the Depth First Search values for a relation's both concerned nodes or queries that retrieve the names of a rule's involved nodes. The results of the products become quite large and reduce the speed of the used algorithms significantly. Although it goes against database design regulations, the problem is solved by adding redundant information to the tables, such that the combinations with the *node* table are avoided; all needed attributes are copied to the involved tables. For example, the table *rules* takes both names of the involved nodes, while the tables *leafrelationi* and *transitiverelationi* take both Depth First Search values of the concerned nodes.

11.4 Implemented database

This section shows the implemented database and explains the not self-explaining attributes. The adjustments to the database as described in the previous section bring about the database scheme depicted on the following page. The scheme contains all the table's attributes that are used in the database.



The part of the implemented database that is able to store all the information that is needed to visualize the include relations of a system.

Some explanation on the non-obvious items:

- The initial structure of each system instance is stored in the attribute *start-structure*
- The *'i'* in the tables *nodei*, *leafrelationi*, *transitiverelationi* and *noderelationi* correspond to the *structureid* of the structure to which the data in these tables belongs
- An attribute *complete* stores whether a structure has been completed. If a user is creating a new structure or is redefining a structure and has not finished/completed it, then the structure should not be available for viewing
- If the structure is a newly introduced structure, then the structure on which it is based is stored in the attribute *originalstructure*
- the values of *viewable* and *transviewable* correspond to whether the users are able to view the direct relations respectively indirect relations in view mode
- The sorting of nodes in the tree-views and the rows and columns of a matrix is done, based on the *childnr* of a node. This *childnr* is attributed to a node according to its position in the alphabetical ordering of its parents children
- If a node's matching structure is a strictly hierarchical structure, then it is assigned to a specific layer. The integer representation of this layer is stored in the attribute *layerdepth*
- If a node's matching structure is a newly introduced structure, then the attribute *orignodeid* stores the *nodeid* of its corresponding node of the original structure. If the node is a structure specific node, then the value of *orignodeid* equals zero
- The attributes of the table *transitiverelationi* match the attributes of the table *leafrelationi*, but has an additional attribute *depth*, which stores the length of the path from *fromid* to *toid* and an attribute *how*, which stores the textual description of the path
- If a structural rule is in conflict with another structural rule, then the textual description of the problem is stored in the attribute *conflict*
- If an architectural rule should not be applied, then the value of its attribute *uses* equals zero, where it would equal one otherwise

11.5 Graph operations

The tool uses two interesting algorithms that calculate the set of cyclic relations that exist in a structure and the transitive closure of the file relations of a system. These two algorithms are examples of graph operations that process a lot of data. The two algorithms are elaborated in this section.

11.5.1 The calculation of cyclic relations

The algorithm is deduced from the algorithm in [7] (p. 554) that calculates the strongly connected components of a graph. A directed graph $G = (V, E)$ is called strongly connected if for every pair of vertices $u \in V$ and $v \in V$ there is a path from u to v and a path from v to u. The strongly connected components of a directed graph are its maximal strongly connected subgraphs, which form a partition of the graph. A cyclic relation is a relation that is a part of a sequence of relations, i.e., a path, from a vertex to itself. Cyclic relations are considered as forbidden relations, because they are in conflict with the general rules of architecture design. The algorithm first finds all strongly connected components. The relations that exist within these components are by definition a part of a cycle. The algorithm calculates the cyclic relations in lineair time $\Theta(V + E)$.

The pseudo code of the algorithm from [7] is given below:

1. call $DFS(G)$ to compute finishing times $f[u]$ for each vertex u
2. compute G^T
3. call $DFS(G^T)$, but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$
4. produce as output the vertices of each tree in the DFS forest formed in point 3 as a separate strongly connected component.

The finishing time $f[v]$ corresponds to the number of completed atomic steps in the algorithm before the vertex v is finalized. G^T is the transposed graph. The set of vertices in G^T equals the set of edges in G, where the set of edges in G^T contains all mirrored edges in G ($\{(v_1, v_2)|(v_2, v_1) \in E\}$).

Subsequently, the algorithm retrieves all the file relations that exist between two vertices that belong to the same strongly connected component. These relations are by definition part of a cycle and therefore violate on the architectural rule that forbids cyclic relations.

11.5.2 The calculation of the transitive closure

Consider a directed graph $G = (V, E)$. The transitive closure of G is a graph $G^* = (V, E^*)$ such that for all $v_1, v_2 \in V$ there is an edge (v_1, v_2) in E^* if and only if there is a non-null path from v_1 to v_2 in G.

The algorithm, which is deduced from [12], uses the results of the algorithm that finds the cyclic relations, namely the calculated set of strongly connected components. These are used to construct the condensation graph $G' = (V', E')$, based on the graph $G = (V, E)$. Each vertex in V' corresponds to a strongly connected component in G and an edge (v_1, v_2), $v_1, v_2 \in V'$ is in E' if and only if there exists an edge in E that connects any of the vertices in v_1 and any of the vertices in v_2.

The algorithm places the vertices of G' in reverse topological order. In graph theory, a topological sort of a directed acyclic graph is a linear ordering of its nodes that is

compatible with the partial order R induced on the nodes where $v_1 \in V'$ comes before $v_2 \in V'$ if there is a directed path from v_1 to v_2 in the graph. The algorithm calculates the transitive closure on the condensation graph G' the following way:

1. for each vertex v_1 in G' in reverse topological order
2. for each vertex v_2 in $Adj[v_1]$
3. if (v_2 not in $Succ(v_1)$)
4. $Succ(v_1) = Succ(v_1) \cup v_2 \cup Succ(v_2)$

$Adj[v_1]$ Is the set $\{v2 \in V' | (\exists e : e \in E' : e = (v_1, v_2))\}$ and, when the algorithm is finished, $Succ(v_1)$ is the set of vertices v_2 for which holds that $v_1 \twoheadrightarrow v_2$, by making use of the edges in E'.

The algorithm subsequently calculates the graph G^* based on the calculated transitive closure of G', namely G'^*.

Chapter 12

Evaluation

The tool has been used to visualize the include relations of four real software systems that came from Philips Medical Systems, Océ and Assembleon, of which the last one granted the disposition of two different systems. The demonstrations were given for the architects that deal with the implementation of their company's coding standards. In order to check this, these architects use the TICS framework regularly.

The architects are well-informed about the structure and behavior of their systems. The demonstrations were observed very attentively, because the tool provided them a new, better overlook of the structure and behavior of the system in terms of the existing include relations.

The visualization metaphor required some time to become comprehend, because the matrix representations are unfamiliar. Once the metaphor became clear, the turmoil started, because there were many things, i.e., dependencies, they could not grasp.

At Océ, everybody launched questions on what caused specific colored cells, because they could not understand why they were there. By retrieving the cause of the colored cells, the information caused reactions like "oeeh!", "aaaaah!" and even red faces.

The demonstration at Assembleon was also well-received, although the matrix did not show all the include relations of their system. Namely, Assembleon makes use of so called IDL (Interactive Data Language) programming, which causes many include relations to be located inside this IDL. Unfortunately, TIOBE Software does not yet extract and store these relations, such that they can be visualized by the tool. Nevertheless, the tool was assessed as a very useful tool in order to draw conclusions on their system, based on the dependencies.

Before the demonstration at Philips Medical Systems was given, their system was carefully looked at, such that possible programming errors or dependencies could

be tracked down. By using the tool, some real errors were actually found by simply browsing their data and by trying to understand their system. As could be concluded from the data, their MR project requires that every building block, or component, may include files that are located within the same building block. If a component includes a file of another component, then the included file must be located in its subdirectory *ext*, which stands for external. However, some of these included files were not located in the *ext* directory, which looked like programming errors or incorrect dependencies.

The data of the MR project was also used to reconstruct their possibly used layering. A layering that consisted out of two layers was constructed by making use of the relation quantities. In spite of the low number of layers, the tool indicated the existence of many other violations on the architectural rules that concern layering.

All demonstrations were evaluated as good, because the tool did give insight in the structure and behavior of their systems. They thought that the tool could be very useful in their pursuit to improve their code, because they can not only use it to browse the data, but also for checking violations on their intended rules. They also saw the tool as a good solution to perform improvement attempts on their systems, because the results become noticeable almost immediately. The most important/relevant extra features that arose during these demonstrations are elaborated in the following section.

The tool forms a whole and can be used as the first release of the tool. Hopefully, the tool will frequently be used and will be useful to improve the value of the inspected systems.

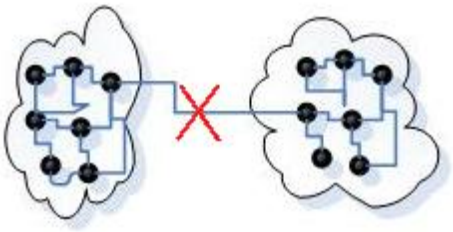
12.1 Future work

The tool described in this thesis fulfils all requirements that were initially claimed by TIOBE Software. However, several other features were brought up during meetings and demonstrations of the tool. This section lists all these features that can be implemented in future projects.

- **A better visualization of components**
  The rows and columns of the matrix representations are sorted on the names of the corresponding vertices of the graph that represents the system's structure. This causes the ordering of the rows and columns of the matrices to be consistent, but it does not increase the users' cognition of the structure and behavior of their system. The visualization metaphor would work a lot better if it would follow the rule "semantic closeness equals visual closeness". This means that the rows and columns should be sorted such that the rows and columns that are similar are visually close together. Most probably, these rows and columns that look the same form a component or are part of the same component. Placing these rows and columns together is very useful for detecting dependencies between components and for making assumptions on the modularity of a system.

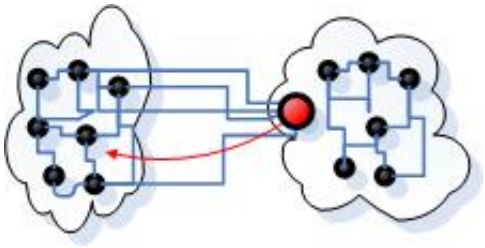
- **Estimate an appropriate layering**
  The tool has the disposition of several sources that can be used to make an estimate for an appropriate layering of a system. These sources are formed by the include relations, the directory structure and the applied rules of a system. By making use of these sources, the tool can compute the probable components of a system. Next, the tool places these components in several layers based on the dependencies between the components. The returned layering is an estimate and can be compared to the layering that (usually) is declared in the design documents. If there is a great difference between the two architectures, then the architects can use this information to find unwanted include relations in the system.

- **Increase system modularity**
  Companies try to obtain a high modularity for their systems, such that the components of these systems can relatively easy be reused. However, some unwanted dependencies may have arisen during the coding phase. This causes the modularity to be lower than planned. For architects it is very tricky to find out where the differences are caused, but by making use of the tool described in this thesis it becomes more trackable. Once the problems are tracked down, they should be solved, such that the modularity increases. The tool could provide aid in this in three different ways. First, the system can suggest to remove certain relations between two components that influence the modularity greatly. The following figure shows a relation that rather should not be there, because it lowers the system's modularity. The tool should therefore suggest that that relation has to be avoided or that it should be removed.



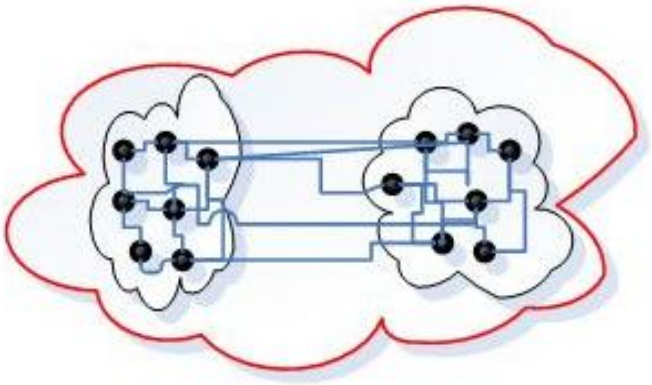
Two components that have a high cohesion. The relation between the two components lowers the system's modularity. The tool should suggest to remove this relation.

Second, the system could suggest to move a part of a component from that component to another component, if this change causes the system's modularity to be increased. The following figure shows such a part that has many relations with another component, while it has no or much less relations with the component in which it is located.



Two components that have a high cohesion. The right component has a part that has many relations with another component, where it has no relations with the component in which it is located. Moving that part from the right component to the left component would increase the system's modularity. This is what the system should suggest.

A third possible change that the tool could suggest concerns the unification of two different components, if the coupling between the components is high. If so, the components can better be combined in one component to increase the system's modularity. The following diagram shows an example in which the tool could suggest to combine the two components.

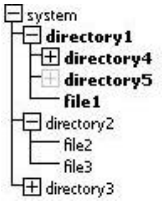


Two components that have a high coupling. The tool should suggest to combine the two components.

- **Generate a system's confidence factor**
  TIOBE Software's framework generates a system's confidence factor that is for example based on the coverage of the system and the weighted sum of violations. This factor is particulary generated for managers, such that they can draw conclusions on their systems based on this factor. The tool described in this thesis can use the information on the include relations and the existing violations to calculate a similar factor that represents how well the system is implemented. This factor can for example be based on the number of violations and the modularity of the system.

- **Direct acces to code**
  From industry came the question whether a specific file relation is connected to the piece of software code that embodies the include statement, such that the including file can be browsed. So, a feature could be implemented to enable users to view the system's code that embodies a selected file relation.

- **Using tree-cuts in the visualization metaphor**
  "A tree-cut is a line crossing each path from the finest grain architectural entities (e.g. B1) to the root (i.e. S) exactly once, where each path is crossed 'through' an architectural entity"[4]. Such a tree-cut can best be described in the following diagram.



This diagram shows a tree-view of a software system. The tree-cut is defined between the visible folded vertices and their parents.

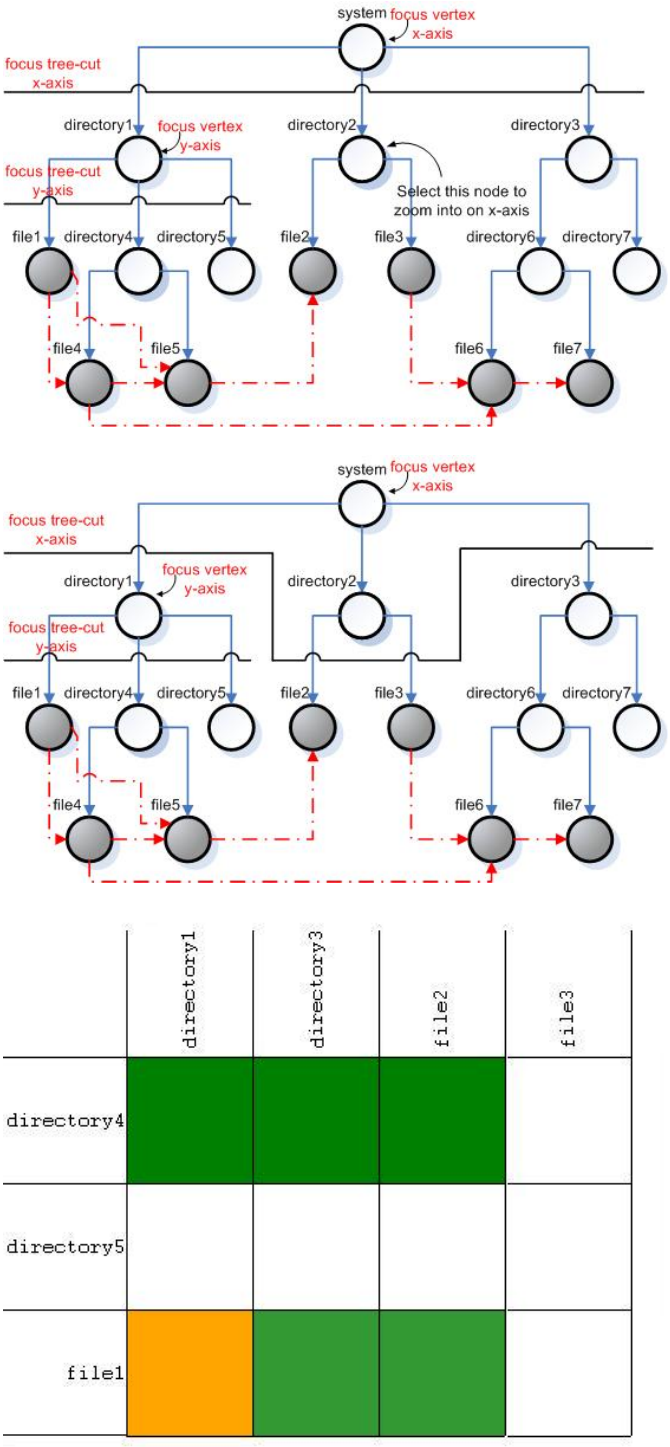
The visualization metaphor enables users to zoom in on a row or column, which results in the change of the corresponding focus vertex. However, as came forth from industry, sometimes it would be useful if the contents of the rows or columns could be defined by these so-called tree-cuts.

By using tree-cuts, it is possible for users to have two different zooming actions on a row or a column: the original zooming action that changes the focus vertex in the corresponding vertex and the new zooming action that allows users to keep all rows or columns the same, except for the selected row or column. This row or column is expanded to a number of rows or columns that equals its number of children.

These two zooming actions require a combination of focus vertices and tree-cuts to define a user's focus. The focus vertices define in which subtree the tree-cut is defined and the tree-cuts are used to indicate which rows and columns should be visualized in the matrix representations, namely the vertices that are positioned below the tree-cut.

An "original" zoom-in action on a row or column causes the matching focus vertex to change in the corresponding vertex of the selected row or column and the matching tree-cut is positioned between this vertex and its children. A "new" zoom-in action on a row or column causes the matching focus vertex to stay the same, but the matching tree-cut is repositioned underneath the corresponding vertex of the selected row or column as visualized in the two figures on the following page.

This extra type of zooming action increases the freedom of the users with respect to the matrix representations. However, it makes the visualization metaphor more difficult to use and it increases the chance of users getting lost. Therefore, an exhaustive research should be performed before this type of zooming is really implemented in the visualization metaphor.

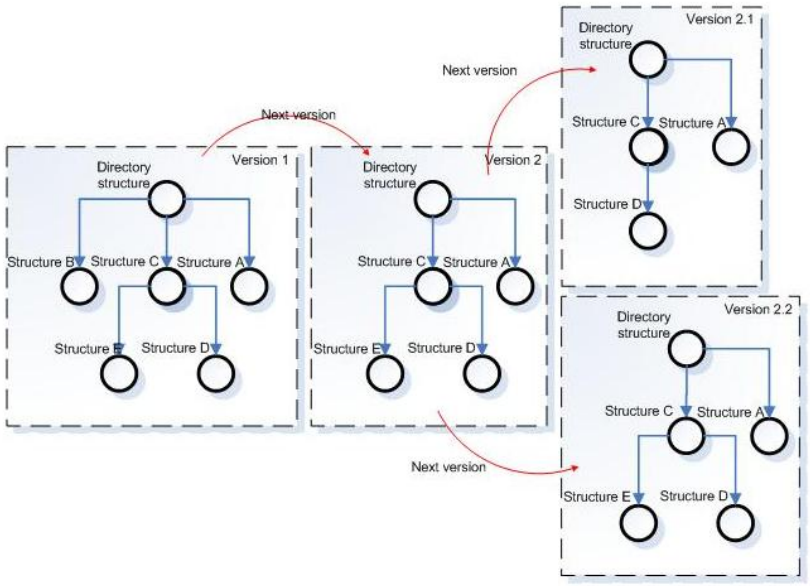


The upper figure shows the current state of the tree-cuts. The selected row is depicted in the figure. The state of the tree-cuts after a zooming action is represented in the middle figure. The lower figure is the resulting matrix representation.

- **Visualize different file relations**
  The tool set can currently be used to visualize include relations in a software system. However, the visualization metaphor can be used to visualize all binary relations between a system's files. Therefore, it would be an improvement if the tool could be used for all sorts of file relations. The user should subsequently be given the opportunity to select the types of relations that he wants to observe. The database is already designed for this, because relations can be given a type and a description. However, the defined rules apply on all types of relations, even though some rules only should apply on a specific type of relations. The rule management part of the tool should therefore be extended in such a way that the users can connect rules with a set of types of relations.

- **Version management**
  The current tool is not able to adequately cope with different versions of a system. To add another version of a system, the user has to create a new instance. Now, different versions of the same system cannot be linked to each other. The versions should be linked to each other so that different functionalities can be implemented to increase the application of the version management. An example of these functionalities is to show the differences between two versions of a system. Another nice feature would be to retrieve a plot of how the value of the confidence factor changes over time.

  The following picture shows how the different structures of a system would relate to each other in a system instance.

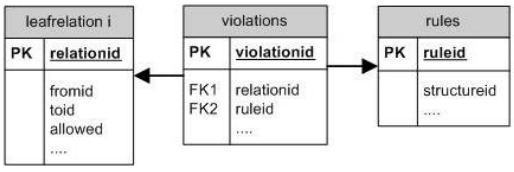


The figure shows the different versions and structures that exist for one system. The rectangles represent versions of the system, where the circles represent the structures of the system. A line between two structures depicts which structure (circle at the head of the relation) is used to define the new structure (circle at the tail of the relation).

- **Integration with TICS framework**
  The TICS framework is the framework that TIOBE Software's customers use to implement, validate and monitor the coding standards that are used to check the quality of their systems. TIOBE Software longs for the integration of the tool described in this thesis, so that their customers can use it to check their system's include relations in an environment that is familiar to them. Currently, the tool needs to be installed separately on the customers' servers, whereas it should be installed together with the TICS framework.

- **Drag and drop in tree-view**
  The user-friendliness of the tool can be much higher. A lack of user-friendliness can be found in the tree-views that are used in the tool. Users now have to press icons to cut, copy, paste and delete selected vertices. The user-friendliness would increase if the tree-view would allow drag and drop functionality.

- **One general set of relations and rules per system**
  The current tool uses a distinct set of relations and rules for each different structure. This means that different structures of the same system make use of a different set of relations and rules. However, the set of relations of a new structure is a subset of the set of relations of the original structure, because the new structure's files are a subset of the original structure's files, and so the connection relations. The rules cause only violations in the structure that they apply to. This is implemented this way because of lack of time to investigate how different structures relate to each other. It would make more sense if all structures of a specific system would work with the same set of relations and that these relations are influenced by all rules defined on all structures of the system. So, if a relation is set to forbidden in a specific structure, then this relation should be visualized as forbidden in all other structures of the same system. However, a user should be enabled to find for each forbidden relation the set of rules that caused this violation. This feature is already implemented in the current tool, but needs adjustment for this feature. A suggestion to implement this is represented in the following diagram.



The figure represents the database tables and their relations that can be used to implement this feature.

The table *leafrelation i* contains the relations that exist in a specific system instance. Here, the *i* represents the identifier of the instance, instead of the structure. A relation is linked to the two files between which the relation exists. The table *violations* contains all violations that exist in the system and forms the link between a forbidden relation and the rules that cause the violations. A relation can be forbidden due to a number of rules and a rule can cause a number of violations.

- **Global and local rules**
  This feature embroiders on the previous possible feature. If all structures work with the same set of relations and rules, then the user should be enabled to declare rules to be local or global, which only affect the structure it applies to, respectively the whole system. Hereby, the users can decide whether a rule should affect relations in other structures or that the rule should only apply to a specific structure.

- **Disallow view mode if it is not up to date**
  If one user changes the structure of a system or changes the rule management part, then the tool disables the view mode for this structure. However, this view mode is only disabled for users that are starting the tool. Users that are already in view mode for that structure can continue in view mode. But because the data for view mode is precalculated, this data does not correspond to the actual state of the structure. A nice feature would be if users are automatically set to editor mode when the view mode becomes disabled, such that users cannot be enabled to work with incorrect data.

Chapter 13

Conclusion

Van Ham already concluded that the visualization metaphor can provide system information in a both effective and conveniently arranged way. A number of demonstrations of Matrixzoom, Van Ham's tool, were given to TIOBE Software's customers. These architects were pleasantly surprised by the provided extra amount of insight in their software products. They expected that the visualization metaphor could be useful to retrieve the differences between the contrived designs and actual implementations of their software systems.

Requirements for the tool were claimed by both TIOBE Software and their customers. The functional requirements regarded some essential conceptual changes, where the non-functional requirements called for technical changes with respect to Van Ham's tool, Matrixzoom.

The tool accepts arbitrary level nestings in a system's structure, such that the users are left more free on the definition on their systems' structures. The visualization metaphor is strengthened by the matrices' independent axes, which enables users also to zoom in on columns and rows. Another important feature concerns design rules. The tool enables its users to check whether and which of their systems' include relations violate on these rules. There are two different rules: structural and architectural rules. The first are user-defined and the second are fixed. The resulting forbidden relations can easily be spotted in the matrices. Furthermore, the tool allows users to manipulate system structures; not only can the users define new structures for a system, but they can also alter an existing structure. These changes can cause changes in the set of forbidden relations, which consequently are visualized in the matrices.

As the demonstrations made clear, the tool gives insight in the structure and behavior of software systems. The architects thought that the tool could be very useful in their pursuit to improve their code, because they can not only use it to browse the data, but also to check violations on their intended design. They also saw the tool as a good solution to perform improvement attempts on their systems, because the results become noticeable almost immediately.

Bibliography

[1] Ham, F.J.J. van - Interactive visualization of large graphs. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 2005.

[2] Ham, F.J.J. van - Using Multilevel Call Matrices in Large Software Projects. In *Proc. IEEE Symp. Information Visualization 2003, IEEE CS Press, pp.227-232,* 2003.

[3] Card S.K., Mackinlay, J. and Shneiderman, B - Readings in Information Visualization: Using Vision to Think. San Francisco, U.S.A., 1999.

[4] Bril, R.J., Feijs, L.M.G., Glas, A., Krikhaar, R.L. and Winter, T. - Hiding expressed using relational algebra with multi-relations - oblique lifting and lowering for unbalanced systems. In *Proc. 4th European Conference on Software Maintenance and Reengineering (CSMR), pages 33-43,* Zurich, Zwitserland, March 2000.

[5] Ball, T. and Eick, S.G. - Software Visualization in the Large. In *Computer, The flagship magazine of the IEEE Computer Society, vol. 29, no. 4, pp. 33-43,* April, 1996

[6] Krikhaar, R.L. - Software Architecture Reconstruction. PhD thesis, University of Amsterdam, The Netherlands, 1999.

[7] Cormen, T.H., Leiserson, C.E., Rivest R.L. and Stein C. - Introduction to algorithms, second edition. The MIT Press, 2001.

[8] Johansson, C. and Wennolf, L. - Software Architecture Verification Tool for Software Platforms. Master thesis, Lund Institute of Technology, Sweden, 2002.

[9] W3C - Scalable Vector Graphics (SVG) Full 1.2 Specification. W3C Working Draft 13 April 2005. http://www.w3.org/TR/SVG12/

[10] W3C - Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation 14 January 2003. http://www.w3.org/TR/SVG/

[11] Wikipedia - De vrije encyclopedie. http://nl.wikipedia.org/

[12] boost C++ libraries - transitive_closure.
http://www.boost.org/libs/graph/doc/transitive_closure.html