MDS-Views: Visualizing Problem Report Data of Large Scale Software using Multidimensional Scaling[*]

Michael Fischer and Harald Gall
Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{fischer,gall}@infosys.tuwien.ac.at

Abstract

Gaining higher level evolutionary information about large software systems is key a in validating past and adjusting future development processes. In this paper we address the visualization of problem reports by taking advantage of the proximity introduced by changes required to fix a problem. Our analyses are based on modification and problem report data representing the system's history. For computation of proximity data we applied a standard technique called multidimensional scaling (MDS). Visualization of feature evolution is enabled through the exploitation of proximity among problem reports. Two different views support the assessment of a system design based on historical data. Our approach uncovers hidden dependencies between features and presents them in easy-to-evaluate visual form. Regions of interest can be selected interactively enabling the assessment of feature evolution on an arbitrary level of detail. A visualization of interwoven features can indicate locations of design erosion in the architectural evolution of a software system.

1. Introduction

Our work is focused on the evolution of software features since they are a natural unit to describe software from the perspective of the application user and the software developer as well. A software design may erode during the project's lifetime and features which were initially implemented in separated modules become more and more interwoven [8]. Our goal is to reveal this degeneration through the use of information collected during the implementation and maintenance phase.

In this paper we describe an analysis and visualization method for uncovering dependencies between sub-modules of software products source tree. Our approach is based on standard techniques and tools and any future project can be adapted easily to meet the data requirements for analysis. Existing projects often have no bug tracking system at all or the integration with the revision control system does not track relevant data.

Modification Reports (MR) and *Problem Reports* (PR) contain an overwhelming amount of information about the reasons for small or large changes to a software system. Groups of reports can be related to provide a clearer picture about the problems concerning a single feature or a set of features. Hidden dependencies of structurally unrelated but over time logically coupled files (i.e. files that most often are changed together although residing in separate modules or subsystems) further exhibit potential to illustrate feature evolution and possible architectural shortcomings.

The approach presented here addresses this problem by grouping feature related PRs and presenting the results in visual form. The input data for this process are selected from a *Release History Database* (RHDB) [12] which contains MR, PR and feature data. For every feature which shall be inspected the related PR information is selected from the RHDB. Now, the distance between two PRs can be expressed as the number of files commonly modified to fix both problems. The more files they have in common the shorter is the distance between the PRs. On the basis of this proximity data, groups of related reports are formed by applying a technique called *Multidimensional Scaling* (MDS) [14] on these data. Results of the MDS process are visualized in a two or optional higher dimensional space. PRs belonging to the same feature are indicated by

[*]This work is partially funded by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT) and the European Commission under EUREKA 2023/ITEA-ip00004 'from Concept to Application in system-Family Engineering (CAFÉ)'.

the same symbol which has been selected to represent the feature. Groups of PRs can be selected interactively and saved for further processing.

Grouping of PRs can reveal hidden dependences between features but can be also used to identify groups of commonly modified program code. Results from this analysis can be used as evidence for a poor system architecture or as indication of design erosion.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of related work in the area of visualization of large large scale software evolution. In Section 3 we introduce the data sources used. Section 4 describes the feature extraction process and its results. Section 5 gives short introduction in grouping using multidimensional scaling. Visualization of problem report data using different views to emphasize different relationships is discussed in Section 6. We conclude in Section 7 with an outlook to future work.

2. Related Work

In [17] Taylor and Munro describe an approach based on revision data to visualize aspects of large software such as active areas, changes made, or sharing of workspace between developers across a project by using animated revision towers and detail views. Since their approach is purely base on revision history, additional important information such as problem reports or feature data are not considered for visualization.

Similar to the our working environment used to produce the results presented in this paper, Draheim and Pekacki propose a framework for accessing and processing revision data via predefined queries and interfaces [9]. Linkage of their data model with other evolutionary project information – such as problem report data as required for our analysis – and making them accessible for external queries is beyond the scope of their work.

Mozilla has been already addressed, for example, by Mockus, Fielding and Herbsleb in a case-study about Open Source Software projects [15]. They also used data from CVS and the Bugzilla bug-tracking system but, in contrast to our work, focused on the overall community and development process such as code contribution, problem reporting, code ownership, and code quality including defect density in final programs, and problem resolution capacity as well.

Bieman et. al [5] used change log information of a small program to detect change-prone classes in object oriented software. The focus was on visualization of classes which experienced frequent changes together which they called *pair change coupling*. Instead of grouping coupled objects they used for visualization standard UML diagram together with a graph showing the number of pair change couplings between change-prone classes.

3. Building a Release History Database

Our analysis is based on three main sources: (a) modification reports (MR); (b) problem reports (PR); and (c) basically the executable program to extract feature information. Data in sufficient quantity and quality is offered via publicly accessible resource from the *Mozilla* project. The MR data are available via the projects Concurrent Versions System (CVS) [7], whereas the PR information is offered via the *Bugzilla* [1] bug tracking system. The source code package for building the executable are released frequently and contain all necessary files to compile the program.

3.1. RHDB

We have downloaded the relevant MR and PR, filtered, validated and stored the these data in our Release History Database (RHDB) [11, 12]. The nucleus of our RHDB is depicted in Figure 1. General information about objects - items or artefacts - of the software's program source and modification reports are stored in *cvsitem* and *cvsitemlog*, respectively. The relation between them was easy to establish, since the relevant information is contained in consistent form within the logs from the CVS repository. Problem reports from the bug tracking system are stored in the *bugreport* entity. Crucial in the reconstruction of the RHDB was the re-establishment of the linkage between modification reports and problem reports since no formal mechanisms are provided by CVS to link this information. In the rebuild process we used the problem report IDs found in the modification reports of CVS to link modification reports and problem reports. These IDs have been entered by the authors of source code modifications as free text. Natural problems were: context not clear in which an ID was used, incorrect report IDs (typos) or no ID at all. Whilst we were able to solve the first two problems with more or less effort, we didn't find a practicable solution for the third problem to reconstruct this information, e.g., from patch data. IDs in modification reports are detected using a set of regular expressions. A match is rated according to the confidence value we have assigned

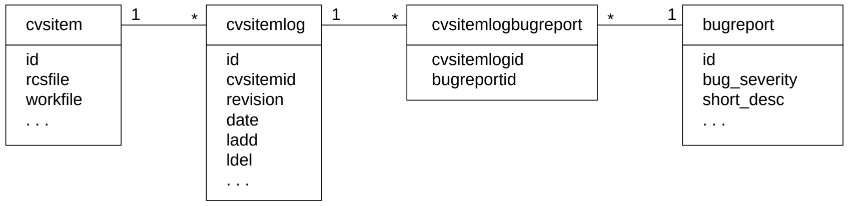


Figure 1. "Nucleus" of the RHDB

to the expression and can be *high* (h), *medium* (m), or *low* (l). The confidence is considered high if expressions such as <*keyword*><*ID*> can be detected whilst a five digit number just appearing somewhere in the text of a modification report without preceding keyword is considered low.

To verify the results from the reconstruction process we used patch information which are sometimes attached to problem reports. These patches contain file name information which can be used to validate the results from the linkage reconstruction. If a patch was found which confirms the linkage, the rating value is changed from (h) to (H), (m) to (M) and (l) to (L), respectively – details can be found in [11].

3.2. Selection of problem reports

To improve results of this analysis process, we need to reduce the impact of PRs not directly related with fixing a specific functional problem. We inspected the description of the largest reports and tried to find a criterion for the selection of our data sets. Reports such as "license foo" (98089,7961), "printfs and console window info needs to be boiled away for release builds" (47207,1135), or "Clean up SDK includes" (166917,888) - the numbers in parenthesis indicate the problem report number and number of referenced files respectively - are considered as not relevant since they primarily concern administrative problems. When the problem reports become smaller they are getting more interesting for the evalution of coupling between features. While the following two reports are still concerned with administrative issues "libtimer_gtk_s is causing link problems" (11159, 300) and "repackage resources into jar files" (18433, 289), the remaining reports begin to focus on programming and bug fix problems: "[meta] necko api revision bugs for embedding" (74221, 254), "[CSS] Rule matching performance improvements" (78695, 234), "Investigate switching output to use DOM Serializer" (50742, 234), "change nsCRT::mem* to mem*" (118135, 233). Thus, we have decided to use 255 as limit for the size of bug reports we accept. This is an arbitrary limit and specific to our configuration. Fortunately, no "major" or "critical" classified PRs are affected by this criterion.

4. Features

Since features are used in communication between users and developers it is important to know which features are affected by (future) functional modifications of a software system. According to [13] a feature is *a prominent or distinctive aspect, quality, or characteristic* [3, 4] of a software system or systems. For our purposes we will refer to the more practical definition of a feature as *an observable and relatively closed behavior or characteristic of a (software) part* [16].

Goal of the feature extraction process is to gain the necessary information to map the abstract concept of features onto a concrete set of files which implement a certain feature. To extract the required feature data we applied the software reconnaissance technique [18, 19] within our *Linux* (*RedHat 8.0* & *SuSE 8.1*) development environment. GNU tools [2] were already used sucessfully in [10] to extract feature data.

We first created a single statically linked version of *Mozilla* with profiling support enabled. From several test-runs where the defined scenarios (see Table 1) were executed, we created the call graph information using the GNU profiler. The call graph information again was used to retrieve all functions and methods visited during the execution of a single scenario. Since our analysis process works on the file level, we mapped function and method names onto this higher abstraction level. In the next processing step, "feature data" were extracted from file name mappings using set operations, e.g., the *Xml* feature using the following expression:

$$\mathrm{Xml} = (\mathrm{MathML} \cap \mathrm{XML}) / (\mathrm{Core} \cup \mathrm{HTTP} \cup \mathrm{PNG} \cup \mathrm{fBlank} \cup \mathrm{hBlank} \cup \mathrm{ChromeGIF}).$$

Table 1. Scenario definitions with features

Scenario	Description	Feature	Color	Files
Core	mozilla start / blank window / stop	Core	White	705
HTTP	TrustCenter.de via HTTP[1]	Http	DeepPink	28
HTTPS	TrusterCenter.de via SSL/HTTP[2]	Https	MediumGreen	6
File	read TrustCenter.de from file	-	-	-
MathML	mathematic in Web pages[3]	MathMlExtension	YellowGreen	13
About	"about:" protocol	About	Gold	3
PNG	sample image[4]	ImagePNG	DarkOrange	10
XML	XML Base[5]	Xml	MediumOrchid	65
JPG	JPEG Karlskirche[6]	ImageJPG	Cyan	16
fBlank	read blank html page from file[7]	Html	DeepSkeyBlue	76
hBlank	blank html page via HTTP[8]	-	-	-
ChromeGIF	Mozilla logo[9]	ImageGIF	SlateBlue1	4
Image	-	Image	OrangeRed1	3

where the names represent the set of files extracted in the previous steps from the executed scenarios. Table 1 also lists the names assigned to the features, the colors which are used later, and the number of files retrieved. E.g., for the *About* feature we determined to following set of files: `content/base/src/nsTextContentChangeData.cpp`, `xpfe/appshell/src/nsAbout.cpp`, and `xpfe/appshell/src/nsAbout.h`. Finally, we imported these filename information into the RHDB along with the release number of the program from which the data were retrieved. In our case it was *Mozilla* version *1.3a* with the official freeze date 2002-12-10.

5. Introduction to multidimensional scaling

The goal of multidimensional scaling (MDS) is to map objects $i = 1, \dots, N$ to points $\|\mathbf{x}_i - \mathbf{x}_j\| \in \mathbb{R}^k$ in such a way that the given dissimilarities $D_{i,j}$ are well-approximated by the distances $\|\mathbf{x}_i - \mathbf{x}_j\|$ whereas k is the dimension of the solution space. MDS is defined in terms of minimization of a cost function called *Stress*, which is simply a measure of lack of fit between dissimilarities $D_{i,j}$ and distances $\|\mathbf{x}_i - \mathbf{x}_j\|$. In its simplest case, *Stress* is a residual sum of squares:

$$\mathrm{Stress}_D(\mathbf{x}_1, \dots, \mathbf{x}_N) = \left(\sum_{i \neq j} (D_{i,j} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2 \right)^{\frac{1}{2}}$$

where the outer square root is just a convenience that gives greater spread to small values [6].

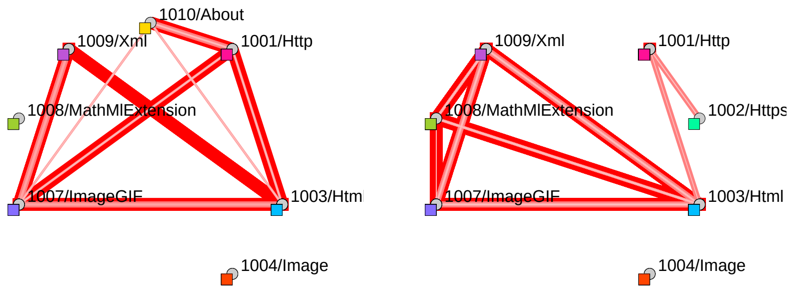
For our experiments we used *metric distance scaling* which is a combination of *Kruskal-Shepard distance scaling* and *metric scaling*. *Kruskal-Shepard distance scaling* is good at achieving compromises in lower dimensions (compared to *classical scaling*) and *metric scaling* uses the actual values of the dissimilarities in contrast to *non-metric scaling* which considers only their ranks [6].

The generation process of the dissimilarity matrix can be formally described as follows. A problem report descriptor d_i of a problem report p_i is built of all artefacts a_n which refer to a particular problem report via their modification reports m_k (linkage MR – PR; see Section 3):

$$d_i = \{a_n | a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i\}.$$

The distance data for every pair of problem report descriptor d_i, d_j are computed according to the formula below and fed

[1]`http://www.trustcenter.de/`
[2]`https://www.trustcenter.de/`
[3]`http://www.w3.org/Math/testsuite/testsuite/General/Math/math3.xml`
[4]`http://www.w3.org/Math/testsuite/testsuite/General/Math/math3.png`
[5]`http://www.w3.org/TR/2001/REC-xmlbase-20010627/Overview.xml`
[6]`http://www.infosys.tuwien.ac.at/img/karlskirche.jpg`
[7]`file:///home/eu/robinson/WWW/blank.html`
[8]`http://intra.infosys.tuwien.ac.at:8092/~robinson/blank.html`
[9]`chrome://global/content/logo.gif`



(a) 1999 (b) 2000

Figure 2. Dependencies between features introduced by large problem reports

---

into the *Dissimilarity Matrix*.

$$\text{dist}(d_i, d_j) = \begin{cases} 1 & \text{if } p_i \not\mathsf{R} p_j, \\ \frac{1}{2}(1 - \frac{n}{\min(s_i, s_j)}) & \text{if } p_i \mathsf{R} p_j \end{cases} \tag{1}$$

where s_i and s_j denote the size of the descriptors d_i and d_j respectively. The fraction $\frac{1}{2}$ is used to emphasize the distance between unrelated objects and “weakly” linked objects. All values are scaled according to the maximum number of elements the descriptors can have in common, i.e., they are scaled to the size of the smaller one.

An alternative way of specifying distances is to use edges and weights. We use a logarithmic function to emphasize the connections with higher values, while connections with lower values are weakened. This has the effect that the nodes with stronger connections are moved closer to each other than nodes with only a few connections. The weight for an edge between two nodes v_i and v_j are computed by the following formula, whereas n specifies the current number of connection between the two nodes and n_{max} the maximum number of connections between two nodes of this graph:

$$\text{weight}(v_i, v_j) = 10 - \lfloor \frac{\ln(n)}{\ln(n_{max})} * 8 + 1.5 \rfloor \tag{2}$$

All weights are mapped by the above formula onto a range of $[1..9]$ where 9 means the closest distance. Other integer values not covered by the given range cause the visualization program *xgvis* [6] to hang or crash. Now we just need to define when two problem reports are linked: p_i and p_j in the RHDB are linked via a software artefact a_n if a modification report m_k exists such that

$$a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i \wedge m_k \mathsf{R} p_j$$

or two modification reports m_k, m_l exist such that

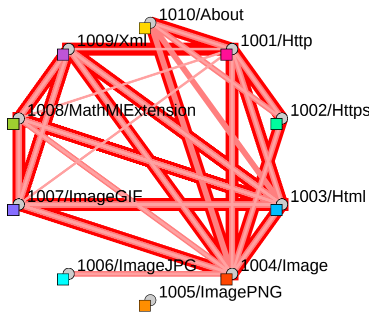
$$a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i \wedge a_n \mathsf{R} m_l \wedge m_l \mathsf{R} p_j.$$

6. Views

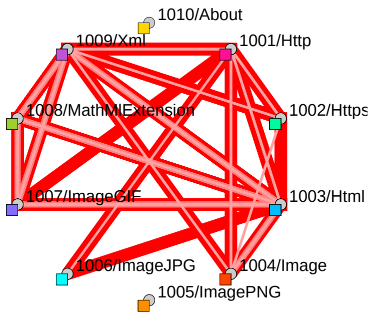
Visualization is a useful technique to present complex interrelationships. We use two different types of views to facilitate the understanding of evolutionary processes in large software: a) *feature-view* focuses on the problem report based coupling between the selected features; and b) *project-view* depicts the reflection of problem reports onto the structure of the project-tree.

6.1. Feature view – projecting PRs onto features

This view focuses on the visualization of features and their dependencies via problem reports. The coupling between two features is symbolized via edges whereas the number of references is expressed as line-width. Since a feature comprises



(a) 2001



(b) 2002

Figure 3. Dependencies between features introduced by large problem reports

several items (i.e., object of the project tree), a connection between two features may consists of several lines of different width. They indicate the coupling of files through problem reports on feature level rather than file level. In fact, all entities contributing to a feature are drawn on the same position, which supports the impression that features are compared.

To reduce the amount of visible edges and to visualize only the most important ones, we used a threshold criterion of 10% of the number of references of the topmost problem report or 20 edges if the number of references is greater than 200. Likewise, in the graphical representation the number of problem reports two features have in common are not depicted. These numbers are listed in Table 2 for all features. This means that every edge in the following figures (except for Figure 2.a) represents at least 20 references. Due to the small set of files we identified for the features *ImageGIF*, *Image*, and *About* and the 10% limit, the coupling for this features may be not very indicative.

Figures 2 and 3 depict the results for the observation periods 1999, 2000, 2001, and 2002, whereas features are aligned on a circle and colored according to Table 1. From 1998 till 1999 – the situation is depicted in 2.a – we found virtually no coupling between features. The situation changed in the subsequent observation periods not dramatically but constantly. In Figure 2.b the situation for the year 2000 is depicted and indicates that the focus shifted from *Http* to other features such as *Html*, *Xml*, and *MathMlExtension*.

Consecutively, the situation changed substantially – as depicted in Figure 3.a – in year 2001. The partially connected graph has turned into an almost fully connected graph and also the number of reported and fixed problems have more than doubled (from 290 to 657). Except feature *ImagePNG*, all features are affected by system-wide changes. In 2002 (see Figure 3.b) the situation improved slightly since the number of reported problems dropped to 580.

6.2. Project view – projecting PRs onto project-tree structure

The goal of this view is the visualization of the reflection of problem reports onto the structure of the project-tree. Our method is to assign weights - to both, the edges of the tree structure and the edges introduced through the coupling of problem reports - and to search for groups in the resulting data set. Output of the optimization process is visualization using a conventional drawing program, whereas the resulting graph is enhanced with feature information and distance data.

6.2.1 General description

Input data for *xgvis* and the drawing program, e.g., *xgvis*, used for visualization are generated by a Java program. This program accepts some arguments which allows to control the data selection and generation process. A critical step in the data generation process is the selection of parameters for the weights since this has a direct impact on the final layout. For edges of the project-tree we use a *weight* of 10. Connections via problem reports are weighted 1 for every connection between two objects - a single report can affect several objects of different directories. This scheme gives more emphasize on the directory structure compared to connections with 1 or 2 problem reports between nodes. In a first phase of the data generation process, the objects of the project tree are assigned the respective nodes of the graph. The minimum child size parameter *minchildsize*

Table 2. Total number of “couplings” between features

	Feature										Period				
Feature	...1	...2	...3	...4	...5	...6	...7	...8	...9	..10	$<$2000	2000	2001	2002	Total
...1/Http	0	20	10	7	0	2	2	0	7	4	24	63	155	129	367
...2/Https		0	2	1	0	0	0	0	2	0	0	1	69	61	130
...3/Html			0	6	0	1	46	16	29	1	87	116	140	122	463
...4/Image				0	0	15	5	0	3	2	5	5	54	31	95
...5/ImagePNG					0	0	0	0	0	0	0	0	3	5	8
...6/ImageJPG						0	0	0	0	0	1	0	22	11	34
...7/ImageGIF							0	17	36	0	15	61	75	38	188
...8/MathMLExtension								0	67	0	2	9	32	72	114
...9/XML									0	1	7	34	105	110	249
..10/About										0	2	1	2	1	6

specifies which nodes remain expanded or will be collapsed. Collapsing means that the object information is moved to the next higher level. We used as default setting a value of 10. To simplify the resulting graph – the complete *Mozilla* project tree consists of more than 2500 subdirectories – we cut off unreferenced directories. With the compactification parameter *compact* it is possible to determine the limit for callapsing directories entries. A limit of 1 means that only unreferenced directories are collapsed/removed. The effect on the graph is that unreferenced leafes are suppressed.

In the following figures the project-tree, i.e., directory structure, is shown as gray nodes connected by black lines. The root node is indicated with the name “ROOT” and features are indicated by colored boxes according to the colors given in Table 1. Coupling between nodes as result of a common problem reports are indicated by pink lines. Broader lines and a darker coloring means that the number of problem reports two nodes have in common is higher. Black lines indicate the structure of the project tree. Since the optimization algorithm tries to place connected nodes close to each other, stronger dependencies can be grasped easily.

One marginal problem is the limited layout area in the two dimensional solution space - all nodes must be placed at least somewhere within a single plane - the placement of nodes after the optimization is only indicative within a certain radius. Naturally, this radius depends on the total number of nodes. By zooming-in, it is possible to give a better picture of otherwise overlayed areas. An n-dimensional solution space could yield better results but is very difficult to visualize. It is also possible that nodes are placed side by side, not because they share a common problem report, rather they have nothing in common with other nodes so they could be attracted by them and moved to a different location. In general, the layout after the optimization is one possible solutions. It also does not mean that a global minimum for the given distances has been achieved.

6.3. Project view results

The initial layout without any optimizationed clustering of the graph for three features – *HTTP* (DeepPink), *HTTPS* (MediumGreen), and *HTML* (DeepSkyBlue) – after data extraction from the RHDB is depicted in Figure 4. Black lines indicate the structure of the project tree, whereas pink lines indicate the coupling via problem reports.

A minimal set of three features - *HTTP* (DeepPink), *HTTPS* (MediumGreen), and *HTML* (DeepSkyBlue) - is depicted in Figure 5. The root of the project tree is indicated by “ROOT”. Easy to see is the placement of HTML features on the right side, and *HTTP*, *HTTPS* on the opposite side. One exception is *.xpcom*: it can be shifted to the left side during the optimization process, but it is automatically moved back to its original position by the optimization program. This means the achieved result is stable and this is an optimal solution for the given weights. Interesting to see is the arrangement of `.security` and `.netwerk` since they were placed close together. In the original layout they were placed in oppositely directions (see Figure 4). What can be deduced from the placement, is the coupling between certain very close together placed nodes such as `.layout.html.base` and `.intl.lwbrk`, or `.netwerk.base` and `.netwerk.protocol.http`.

Figure 6 depicts all features we extracted from *Mozilla*. The root of the project tree has been shifted away from the center and its structure has been completely mangled by the optimization procedure on the basis of the given weights. As the previous figure already shows, it is also possible to assign distinct areas for the features, e.g., the *Http* and *Https* area by the colors DeepPink and MediumGreen. Features in other areas are not that distinguishable as desired.

Figure 7 depicts a detailed view of the cluttered area of Figure 6 with its coupling between different levels of the project structure. The positions of the nodes are slightly modified to have a better “viewing position” on the coupling. Since the coupling lines connect nodes of the project tree not along the project tree path, it may indicate unstable interfaces or

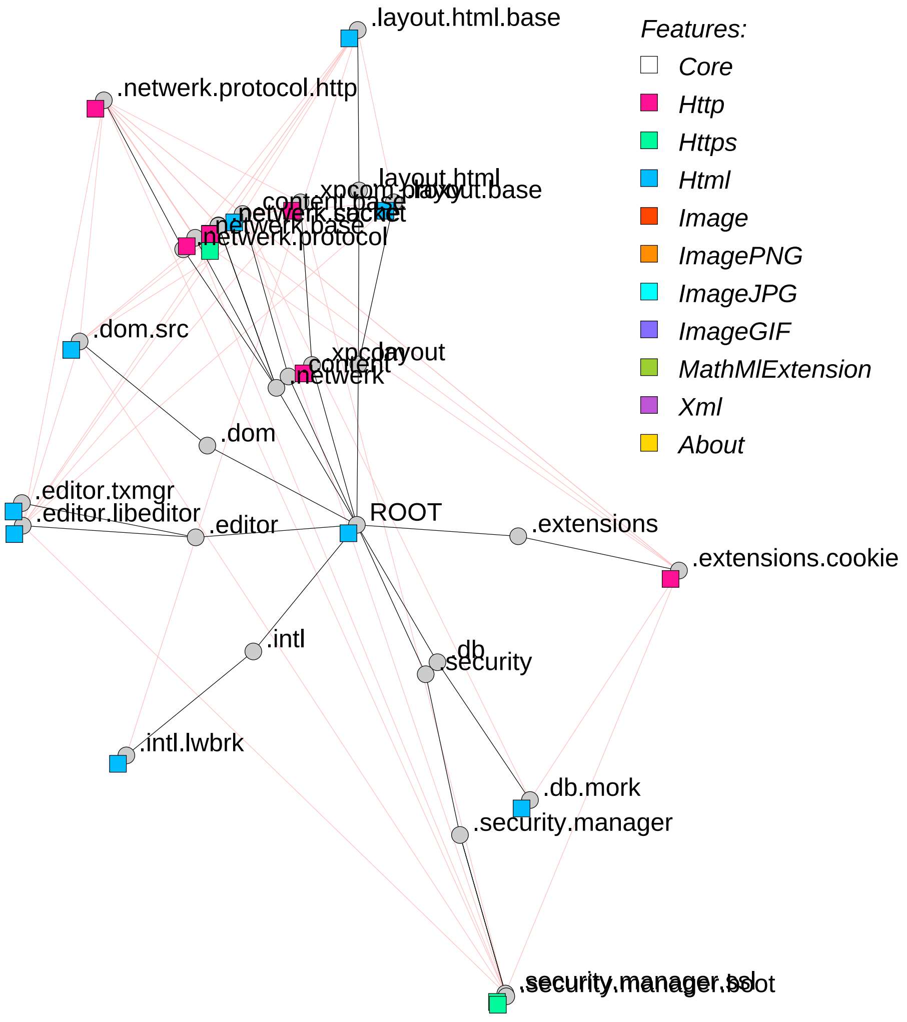


Figure 4. Initial layout for features: Http, Https, Html

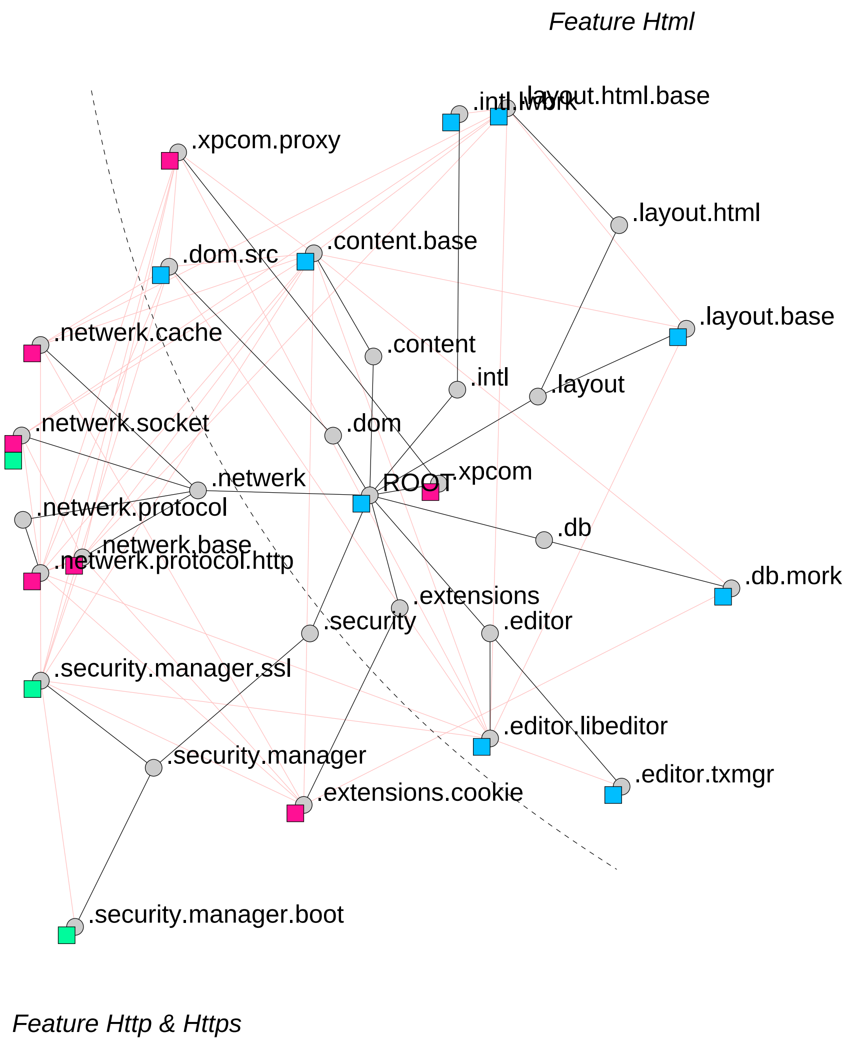


Figure 5. Features: Http, Https, Html

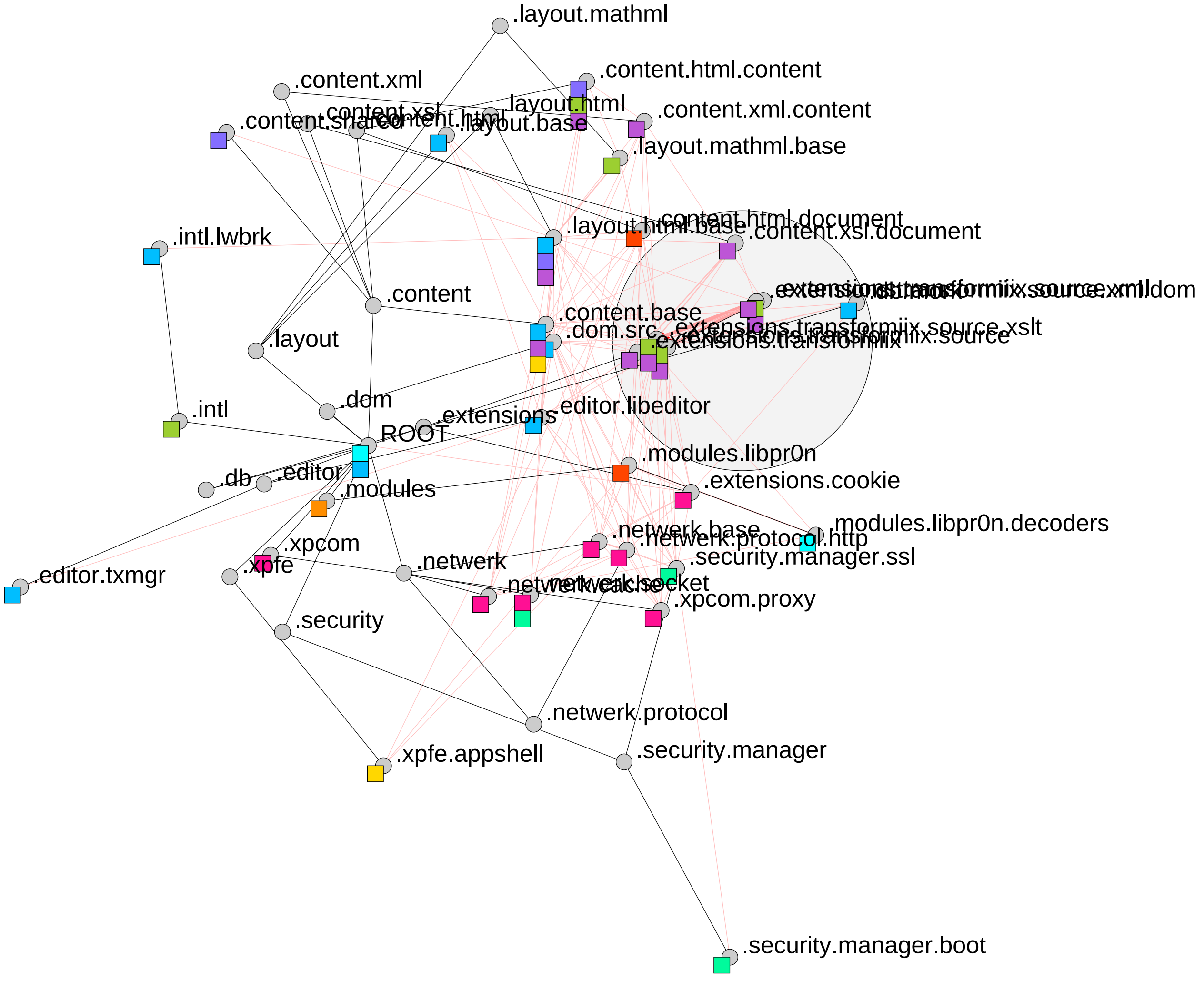


Figure 6. All features but no core

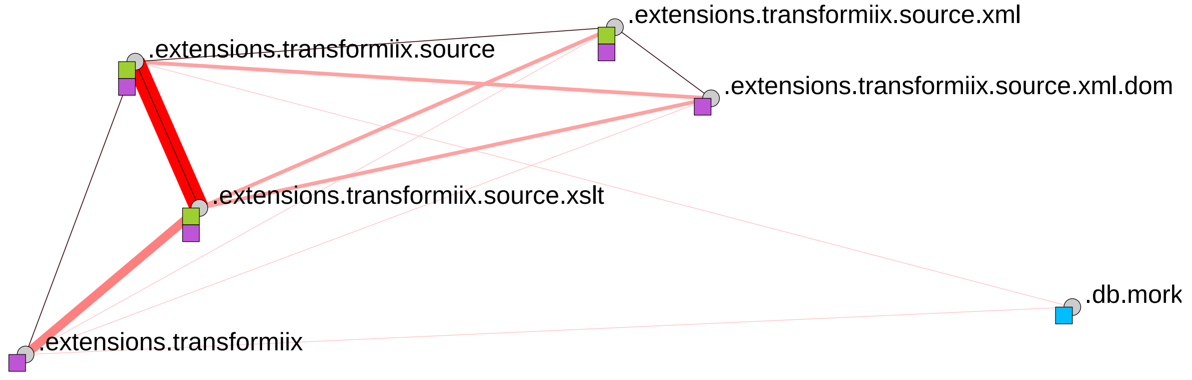

.extensions.transformiix.source.xml
.extensions.transformiix.source
.extensions.transformiix.source.xml.dom
.extensions.transformiix.source.xslt
.db.mork
.extensions.transformiix

Figure 7. Detailed view of Figure 6

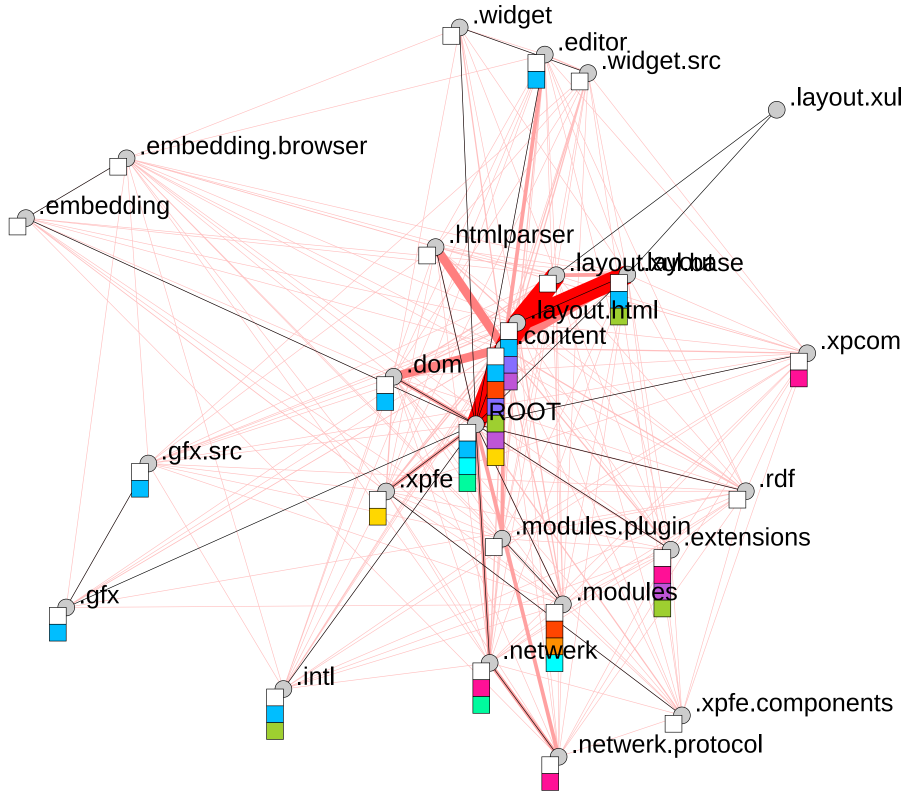


Figure 8. Core and features

weaknesses in the system design.

The core - indicated by white boxes - and all features are depicted in Figure 8 on a very coarse level. For this configuration we selected all reports which were rated *major* or *critical*. We also set the minimum sub-tree size to 250 (*minchildsize*) entities and the minimum number of problem report references to 50 (*compact*). As result we received a graph with 25 nodes, 215 edges induced by problem reports. By changing the values for *minchildsize* and *compact* it is possible to generate an arbitrary detailed graph of the while project. Since *Mozilla* has more than 2500 sub-directories a complete graph representation of the whole project tree is far beyond the illustration capabilities of this medium. It is intuitive that the most critical sub-systems in *Mozilla* are related with visualization which is also supported by our findings. The nodes with the highest density in severe problem reports are `.content` (with 595 references), `.layout.html` (438), `ROOT` (366), `.layout.xul.base` (220), and `.layout` (210). In fact, `ROOT` does not have any entities, but through the compactification of the project tree entities from lower levels have been moved to higher levels. The same is true for the other nodes such as `.com` or `.htmlparser`, since all the entities are moved to higher levels until the *compact* criterion is met. Nodes with fewer connections are `.dom` (161), `.xpfe` (121), `.netwerk` (119), `.modules` (118), `.modules.plugin` (106), `.network.protocol` (100), `.rdf` (100), `.htmlparser` (96). Another interesting aspect is the spread of edges. In total 215 connections between nodes are depicted. While some nodes are more focused on a single partner node, e.g., `.htmlparser` with 15 edges, others have a wider spread, e.g., `.xpfe` with 21 edges. Since `.layout.xul` does not share a problem with any other node, we can neglect this node and have 24 nodes sharing connections with others. For instance, `.xpfe` shares connections with 87.5%, or `.content` with 95.8% of the possible nodes.

7. Conclusions and future work

The graphical representation of dependencies between features based on problem report data opens a new perspective on the evolution of software systems through retrospective analysis by visualization. By intuition problem reports should have a minimal impact on different features. Situations where this is not the case can be grasped easily through the graphical representation, e.g., in case of overlapping or feature spreading. We have applied multi-dimensional scaling of problem reports linked with files and directory structures for the visualization of features of Mozilla for the years 1999 until 2002. The tool that we developed allows a domain expert to generate two specific views of relationships and dependencies of a large software system: (1) the *feature view* enables a projection of problem reports onto the files that realize a particular feature, thereby indicating otherwise hidden feature dependencies that have evolved over time (intentionally or unintentionally); (2) the *project view* enables a projection of problem reports onto the directory and project structure of a system and, as a result, depicts the logical coupling between modules, sub-modules, etc. introduced through changes over time.

First results using MDS are promising, thus we want to further explore this approach and test other large software systems to compare, for instance, the spread of features in commercial and other open source software. Of further interest are the exploration of higher dimensional solution spaces which should yield more optimized solutions. With *xgvis* this is quite difficult since the interactive selection and visualization works optimal only on two-dimensional data.

An interesting perspective for future work is the coupling of this visualization approach with architecture recovery systems. One possible application could be to gain insight into the impact of problem reports on architectural styles and patterns. A pattern search process might identify all implementations of a socket connection. The location information is augmented with information from the RHDB and visualized using MDS. More work will be devoted to visualization capabilities such as highlighting or selection of diverse areas for detailed inspection of problem reports. Furthermore, we will investigate the optimization algorithms as to allow placing related features as close to each other as their proportional computed strength indicates.

8. Acknowledgments

We thank the *Mozilla* developers for providing all their data for this case study to analyze the evolution of an large Open Source Software project.

References

[1] Bugzilla Bug Tracking System.
`http://www.bugzilla.org/`.

[2] GNU's Not Unix! - the GNU Project and the Free Software Foundation (FSF). `http://www.gnu.org/`.

[3] *Merriam Webster's Collegiate Dictionary*. Merriam-Webster, Incorporated, 10th edition edition, 1996.

[4] *The American Heritage Dictionary of the English Language*. Houghton Mifflin Co, 4th edition edition, 2000.

[5] J. Bieman, A. Andrews, and H. Yang. Understanding change-proneness in OO software through visualization. In *Proceedings of 11th International Workshop on Program Comprehension*. IEEE, 2003.

[6] A. Buja, D. F. Swayne, M. Littman, N. Dean, and H. Hofmann. XGvis: Interactive Data Visualization with Multidimensional Scaling. *Tentatively accepted for publication in the Journal of Computational and Graphical Statistics*, 2001. `http://www.research.att.com/areas/stat/xgobi/papers/xgvis.pdf`.

[7] P. Cederqvist et al. *Version Management with CVS*, 1992. `http://www.cvshome.org/docs/manual/`.

[8] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[9] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE), Helsinki, Finland*. IEEE Computer Society Press, September 2003.

[10] T. Eisenbarth, R. Koschke, and D. Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, November 2001.

[11] M. Fischer, M. Pinzger, and H. Gall. Analyzing and Relating Bug Report Data for Feature Tracking. In *10th Working Conference on Reverse Engineering (WCRE), Victoria, Canada*, November 2003.

[12] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 2003 International Conference on Software Maintenance (ICSM 2003), Amsterdam, Netherlands*, September 2003.

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda). Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[14] J. B. Kruskal and M. Wish. Multidimensional Scaling. *Quantitative Applications in the Social Sciences*, 11, 1978.

[15] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.

[16] E. Pulvermüller, A. Speck, J. O. Coplien, M. D'Hondt, and W. DeMeuter. Feature Interaction in Composed Systems. In *Feature Interaction in Composed System*, 2001.

[17] C. M. B. Taylor and M. Munro. Revision towers. In *1st International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2002.

[18] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *International Conference on Software Maintenance*, pages 200–205, 1992.

[19] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, January 1995.