Case Study: Visual Analytics in Software Product Assessments

Alexandru Telea*
Institute for Math. and Computer Science
University of Groningen, the Netherlands

Lucian Voinea†
SolidSource BV
Eindhoven, the Netherlands

Abstract

We present how a combination of static source code analysis, repository analysis, and visualization techniques has been used to effectively get and communicate insight in the development and project management problems of a large industrial code base. This study is an example of how visual analytics can be effectively applied to answer maintenance questions and support decision making in the software industry. We comment on the relevant findings during the study both in terms of used technique and applied methodology and outline the favorable factors that were essential in making this type of assessment successful within tight time and budget constraints.

1 Introduction

Industrial software projects encounter bottlenecks due to many factors: improper architectures, exploding code size, bad coding style, suboptimal team structure, or mis-estimation of the requirements. Understanding the causes of such problems helps taking corrective measures for ongoing projects or choosing better development and management strategies for new projects.

In research contexts, software analysis techniques such as static and dynamic program analysis and software visualization are frequently advocated as effective in getting insight into large software systems. Many visualization tools have been proposed to this end, an overview of which is given in [Diehl 2007; Spence 2006; Ware 2004]. However, there is a tendency in some visualization papers to focus on presenting novel techniques and tools, rather than describing in detail how these tools have actually been used to answer concrete questions in the software industry. Some prominent researchers have voiced concerns on the development of such trends in the fields of software visualization and information visualization. They pointed out to a strong need for work that explains how research-grade tools and methods are actually used in answering such concrete questions, and the need for low-cost, shallow learning-curve solutions that can be (and are) adopted by the industry [Reiss 2005; Koschke 2003; Lorensen 2004].

We present here such a case study that describes how a set of visualization and static data analysis tools was used to understand the causes of development and maintenance problems in an industrial project. Rather than proposing novel visual metaphors, we propose novel combinations of existing visualization and static analysis methods to answer a number of concrete questions from the management team of a large software project. The uncovered facts helped the team leadership to understand the causes of past problems, rule out false suppositions and validate earlier suspicions, and assisted them in deciding further project course. Our solution fits into the emerging *visual analytics* discipline [Wong and Thomas 2004], as it uses information visualization to support the analytical reasoning about data mined from code repositories.

Section 2 gives a short description of the premises of our analysis and the goals of the stakeholders. Section 3 outlines the analysis method used and the constraints we had to face, and details on the several types of analyses done: metrics, modification requests, team identities, dependencies, code duplication, and documentation. Section 4 details the obtained results, involving visual analysis of the repository's modification requests, code metrics evolution, code dependencies and duplication, and documentation. Section 5 discusses the results from both the stakeholders' and the analysts' perspective. Section 6 concludes the paper.

2 Software Project Description

The studied software was developed in the embedded industry by three teams located in Western Europe, Eastern Europe, and Asia[1]. All code is written in Keil C166, a special C dialect with constructs that closely supports hardware-related operations [Keil, Inc. 2008]. The development took six years (2002-2008) and yielded 3.5 MLOC of source code (1881 files) and 1 MLOC of headers (2454 files) in 15 releases. In the last 2 years, it was increasingly felt that the project could not be completed on schedule, within budget, and that new features were very hard to introduce. The team leaders were not sure what went wrong, so an investigation was performed at the end (2008).

The main questions raised were: why the project went beyond schedule (was it because of bad architecture, design, or management); why it was hard to add new features to the existing software; and how a follow-up should continue (start from scratch or continue with the existing code). As findings were obtained, these questions were further refined, as described in the following.

3 Analysis Method

The investigation, performed by the authors, had two parts: a process analysis and a product (code) analysis. We describe here only the latter.

3.1 Tooling

The only input for the analysis was the source code and documentation, stored in a Source Control Management (SCM) system. First, we performed static analysis on each version of each source file, using a heavyweight C/C++ analyzer able to handle incorrect and incomplete code by using a flexible GLR grammar [Telea and Voinea 2008a]. We were able to easily modify this grammar (within two work days) to make it accept the Kyle C dialect. The analyzer creates a fact database that contains static constructs, *e.g.* functions and dependencies, and several software metrics, for each version of each analyzed file. We applied the above process to all files in all 15 releases. Given the high speed of the analyzer, the entire static analysis took under 10 minutes on a standard Linux PC.

*e-mail:a.c.telea@rug.nl
†lucian.voinea@solidsource.nl

[1]The exact details of the project have been removed to preserve anonymity.

Next, we visualized the trends (evolution) in both the analyzed data and in the modification requests (which couple the requirements with their actual implementation), commits, and authors information available in the SCM system. For this, we used an implementation of dense-pixel evolution visualization proposed in [Voinea and Telea 2007]. This tool provides both mechanisms to extract such information from the SCM system, as well as several visualizations such as two-dimensional evolution views, treemaps, table lens views [Telea 2006], and classical time series graphing. Each visualization looks at different aspects of the code, so their combination aims to correlate these aspects into one coherent decision-making conclusion.

3.2 Constraints

Several constraints existed during this analysis process. First and foremost, the entire process, from data acquisition to findings interpretation, was limited to a few days, both for cost and efficiency reasons, so there was very little room for experimentation. Secondly, the analyzed software was entirely unknown (both as structure and functionality) to the researchers doing the analysis. Third, the actual customers were not familiar with any of the visualization metaphors used. Fourth, the questions (as can be seen from above) were relatively high-level and involved multiple process and product parameters. Last but not least, the results had to be presented in an understandable form both to domain specialists (architects) as well as the management team, as consensus on the findings and way to go had to be built.

The visualizations, presented in the next section, were interactively shown to the stakeholders, who were asked to comment on the findings, their relevance, and their usefulness for understanding the actual causes of the development problems. During these discussions, the main questions were refined by the customers and sub-questions were answered, during the discussions, based on the obtained analysis and visualization results. The entire visual analysis session took under five hours (see Sec. 5).

4 Analysis Results

In this section, we describe a selected subset from the different types of visual analyses performed on the provided code base. We focused here on those analyses which are the most relevant for the points made in this paper, and which can be described without entering in details about particularities of the analyzed software.

4.1 Modification Request Analysis

Figure 1 shows two images of the project structure, depicted as a three-level treemap (packages, modules and files). Treemaps have been used in many software visualization applications to show metrics over structure [Balzer and Deussen 2005]. The smallest rectangles are files, colored by various metrics, and scaled by file size (LOC)[2]. The top image shows the number of modification requests (MRs) per file. Files with more than 30 MRs (red) appear spread over many packages. The bottom image shows the same structure, colored by team identity. We see that most high-MR files (red in the top image) are developed by Team A (red in the bottom image), which is located in Asia. Seeing this image, the customer realized that this was a potential problem cause, as it was known that this team had communication issues with the other two teams, located in Europe. A proposed better work division was to reassign Team A to work on a single, low-MR-rate, package, or at least to shift the high-MR packages away from that team.

We further analyzed the MR records. Figure 2 left shows the MR distribution over project and time for a subset of files in a package of interest. Files are drawn as dark gray horizontal pixel lines,

[2]We recommend viewing this paper in full color

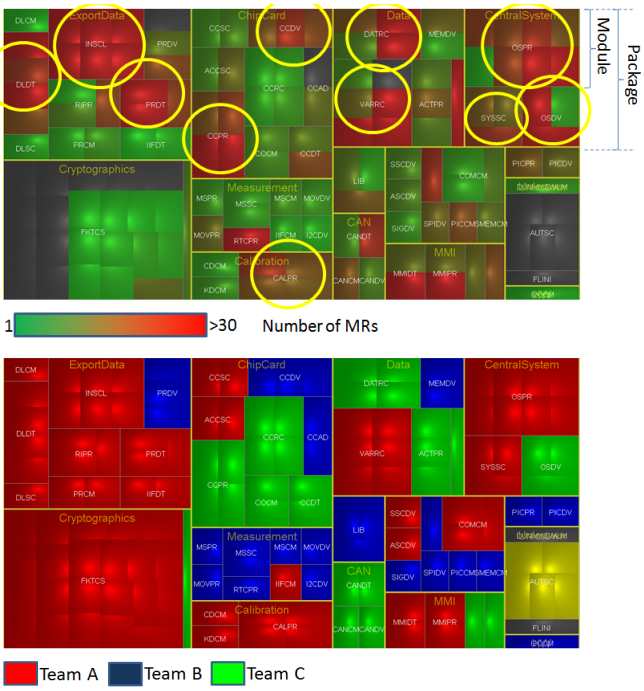

Figure 1: Team assessment: number of MRs (top), with high-activity modules marked, and team structure (bottom)

stacked in creation order from bottom to top. The x axis shows time (years 2002 to 2008), following the visualization model in [Voinea and Telea 2006]. Red dots show the location (file and time) of the MRs. We see that less than 15% of the files have been created in the second project half, so code size explosion could not have been the cause of the project's failure to complete on time.

However, we also see that most MRs in this second half address *older* files, so the late work mainly tried to fix, or satisfy, old requirements. Stronger, the MRs in the last year, outlined in yellow in the image, where activity was quite intense, address for their greatest majority files created more than three years ago. The right image supports this hypothesis: each horizontal bar indicates, with a graph, the number of file changes related to one MR range. Here, MRs are functionally grouped per range (100 MRs per range), where the oldest ones, shown at the top, have the lowest IDs and the newest, shown at the bottom, the highest IDs. The x axis shows again time. We see that older MRs have large activity spreads over time. For example, in mid-2008, developers still try to address MRs introduced in 2005-2006 (!). Clearly, new features are hard to introduce if most work has to deal with old MRs. This correlates positively with the suboptimal team communication and assignment, which leads to increased average MR closure time, *i.e.* time from the first to the last file change related to an MR. Ideally, we would like to see in this image a band-like structure, *i.e.* that all MRs are definitively closed after a given time interval.

Figure 3 shows the average closure time of an MR. This image also correlates with Figure 1. That is, critical MRs, involving Team A, took quite long to close. This partially explains the encountered delays and further supports the idea of reassigning critical MRs to other teams.

4.2 Code Metric Analysis

We next analyzed the evolution in time of several software metrics (see Fig. 4). Although more metrics were computed such as package cohesion, comment density, and usage of potentially dangerous code constructs, we focus here on the simpler ones (fan-in, fan-out,

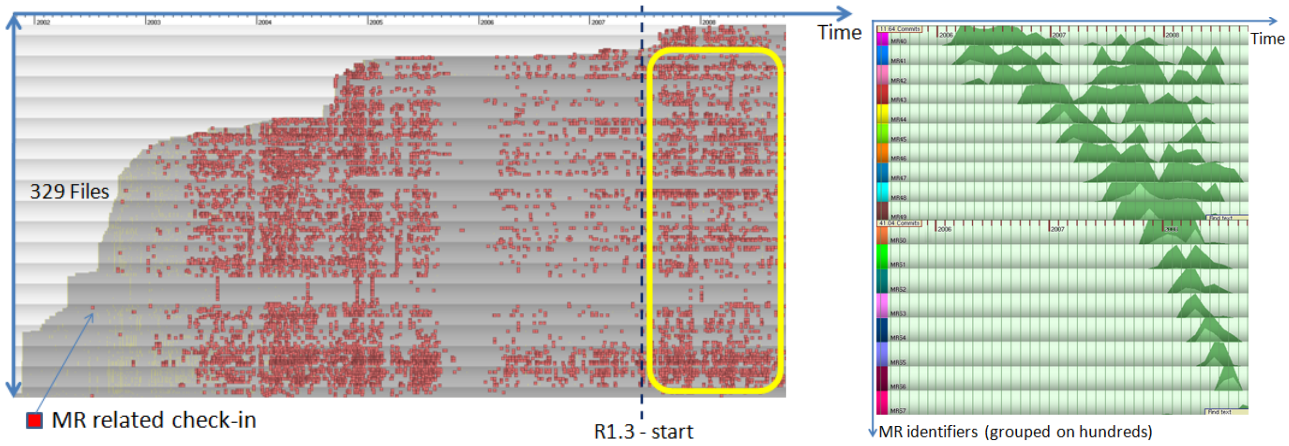

Figure 2: MR evolution per file (left) and per range of MRs (right)

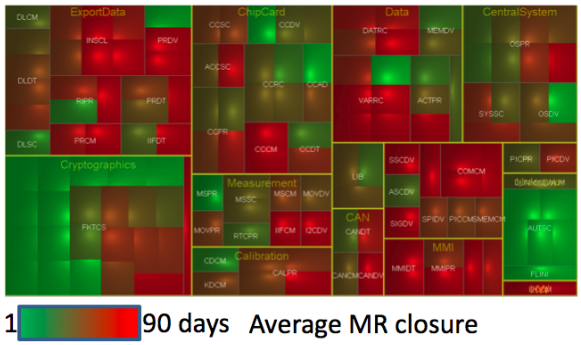

Figure 3: Average MR closure time (red: over 90 days)

functions and function calls counts, and average and total cyclomatic complexity) as these were easier to interpret and supported answering the relevant questions. These metrics were also included as developers mentioned that understanding, changing, and testing existing code was quite hard. The question was whether this was due to a bad code structuring or other (*e.g.* team-related) factors.

The graphs show a relatively low increase of project size (functions and function calls) and, roughly, and overall stable dependency count (fan-in) and code complexity, especially in the second half of the project lifetime. The fan-out and number of function calls increases more visibly. This supports our supposition that maintenance problems were not caused by an increase in code size or complexity, as is the case in other projects. The sharp jumps of complexity and fan-out visible around the first third of the project are correlated with the sharp increase of code in the same period (Fig. 2 left) and also high documentation check-in in the same period (see Fig. 10, further discussed in Sec. 4.5).

However, we see in Fig 4 that the average cyclomatic complexity per function is very high (over 20). It is well known that such values of complexity correspond to code that is hard to maintain and/or test. This led us to further analyze the distribution of metrics over the project structure. Figure 6 shows this distribution, again using treemaps colored by metric values. Comparing this with Figure 1, we see that the modules having the largest number of lines of text, actual lines of code, and complexity are also those with the highest MR counts. This further enhanced the conviction of the stakeholders that the most complex MRs, which arguably caused the largest and most complex implementations, were assigned incorrectly to the suboptimal team. Moreover, the fact that the complexity metric was high from relatively early in the project, and next stayed constant, supported the explanation that long MR closure times were due to early suboptimal development patterns, which next needed a

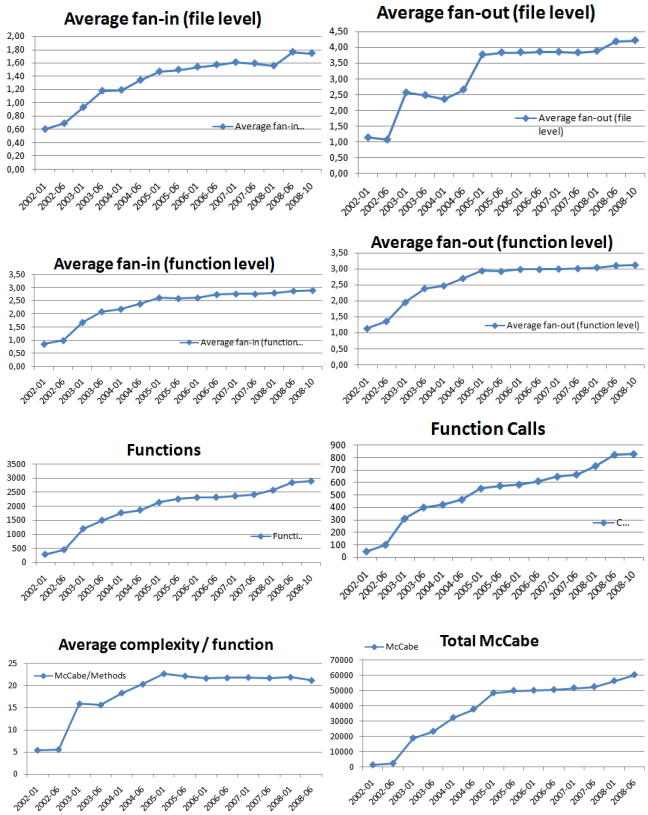

Figure 4: Evolution of static analysis metrics

lot of fixes and work.

The graphs discussed above are useful to have a global overview of how one, or several, metrics evolve in time over the entire software system. However, after getting this type of global insight, it is often useful to restrict the comparison to a subset of modules. For example, when studying a complexity metric, it is useful to see how the complexity evolved in time *and* which are the subsystems which are (increasingly) highly complex. To facilitate this analysis, we propose a combination of treemaps and graphs (see Fig. 5), along the lines of the enhanced treemap technique discussed in [Telea 2006]. The treemap shows the system structure, much like in Fig. 1. For illustration variation, we now also show function definitions within files (these are the treemap leaves) and we use a strip layout as opposed to the squarified layout used before. Atop of each module, we draw the graph of the evolution in time of a selected metric of

interest, computed on the artifacts in that module. This allows us to compare the evolution of that metric over different parts of the system.

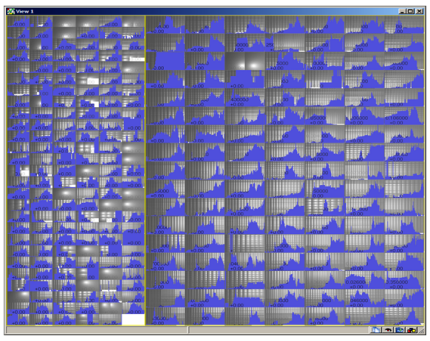

Figure 5: Metric evolution graphs correlated with system structure

4.3 Dependency Analysis

As the stakeholders settled with the conclusion that the project problems were early created by the improper team assignment, the next question was whether the latest version of the code could be refactored to continue development in a new project with different teams. To assess this, we extracted a number of dependencies using the C/C++ analyzer. We visualized these using the bundled edge method proposed in [Holten 2006], which shows hierarchy relations (file in module in package) as a set of nested concentric rings, each ring sector being a software element, and dependencies as curves grouped according to the hierarchy.

Figure 7 shows the inter-file dependencies - more precisely, dependencies between source (implementation) files. Here, we consider that a file A depends on a file B if it directly uses any kind of symbol declared by B. In C, these are function, type, and `extern` variable declarations, as well as macros. We considered all these dependencies, and not just usage of function prototypes declared in headers, since modification of any of these symbols in a public interface would cause ripple effects into the clients of the interface, as the code makes extensive use of macros and typedefs from headers throughout the implementation. Defining dependencies differently is very easy, as the C/C++ analyzer provides complete AST and semantic (type) information on the input code [Telea and Voinea 2008b; Telea and Voinea 2008a]. Getting this information with other C static analyzers, such as CScout [Spinellis 2008], is more difficult, due to the particular C dialect used in this project.

Analyzing this image reveals whether the overall product is modular. The left image shows all file-to-file dependencies. Edges are colored dark red at files providing symbols and light red at files using (referring to) symbols. Due to the bundling, we quickly discover three dominant, concentrated interface providers: the packages *interface*, *plaform*, and *basicfunctions*. These contain public interfaces, so dependencies towards them is allowed. The right image shows all dependencies after the allowed ones, mentioned above, have been filtered away. To make this image easier to use, we filtered away modules that did not have any undesired dependencies - in other words, we zoomed in on that subset of modules that exhibits undesired dependencies. We are now left with a small, though significant, set of unwanted dependencies. These include, among others, references to *extern* global variables or macros not placed in the public interfaces. The conclusion drawn from this was that refactoring by replacing problem modules should be relatively easy from an interface perspective, as there are relatively few unwanted dependencies, and we can see where they are.

Next, we analyzed the function call graph (Fig. 8) The left image shows a quite complex call pattern. We see few intra-package calls

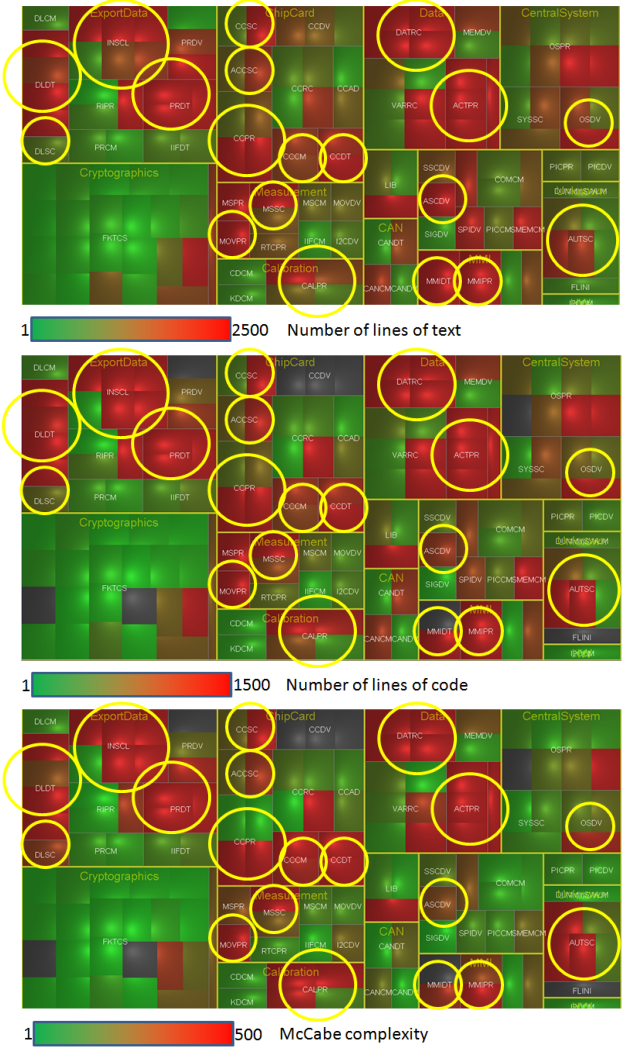

Figure 6: Distribution of text size, lines of code, and complexity over structure. Modules with high values are marked.

and many inter-package calls. This indicates that it would be hard to test the system during a potential refactoring, as all implementations (modules) need to be in place for the whole to work. To further determine if incremental refactoring would be easy, we produced the right image, which shows only those modules which are mutually call-dependent. Such modules have to be refactored as a whole. The involved architects also quickly noticed, on the mutual call dependency image, a number of previously unknown violations, *i.e.* mutually dependent modules that were supposed to be part of a layered architecture with a strict top-to-bottom dependency. Similar insights can be obtained with different visualizations, *e.g.* the matrix view in [Abello and van Ham 2004]. However, from our past experiences, where we used both the matrix view and bundled edge view, we noticed the latter to be much easier to understand by software engineers than the former - possibly due to their familiarity with node-link visual representations. The fact this view was easy to understand, in a matter of minutes, also proved true in the current project.

Overall, the dependency analysis results were interpreted by the stakeholders as follows: the system had an originally clean design, consisting of a layered architecture and a clearly defined small set of public interfaces. As the project progressed, this architecture degraded. Rapid 'hacks' were introduced to fix important change requests, yielding undesired dependencies not taking place via the public interfaces, and also mutual dependencies violating the layer-

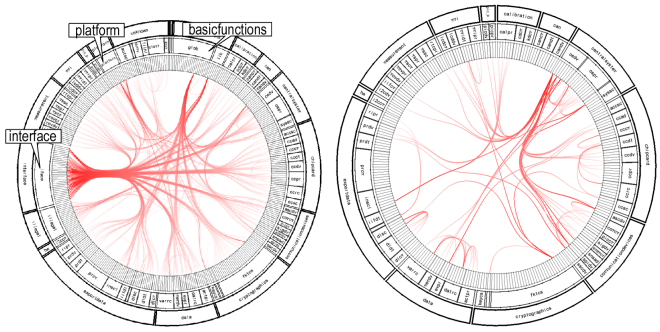

Figure 7: Dependency graphs: complete set (left); subset of undesirable dependencies (right)

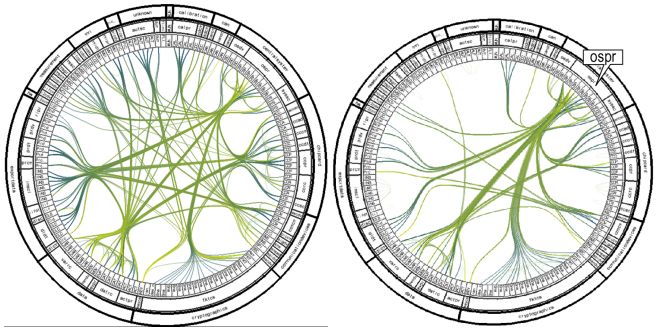

Figure 8: Call graphs: complete system (left); mutual dependencies only (right)

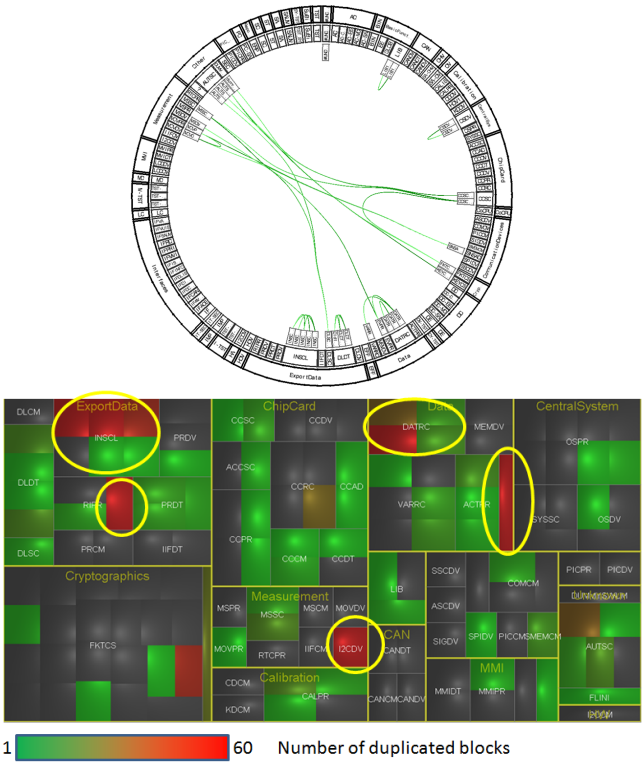

Figure 9: Code duplication: external (top) and internal (bottom). Duplication appears to be minimal.

ing principle. The 'hack' hypothesis was checked by looking at the names of the modules exhibiting dependency problems, and associating them with the teams who worked on those parts of the code and with prior knowledge on how and what these teams performed. This is one of the several examples stressing the need to active involve the stakeholders in the analysis, as such information would not have been available to an external party.

4.4 Code duplication analysis

At this point of the overall study, the consensus appeared that an iterative refactoring would be doable, though not easy. This process would inevitably have to involve the implementations of all modules containing, *e.g.*, violations of the layered architecture or usage of non-public interfaces. To further assess the risks of such a proposal, we next performed a code duplication analysis. Specifically, we looked for duplicate code blocks both within the same file (internal duplication) and duplicate blocks across files (external duplication). Identifying such blocks and correlating them with the previous findings can help in several ways. First, if a duplicated block is modified during refactoring, then we should examine if its duplicates for refactoring too. Second, code modifications caused by insight found during testing and debugging should be done consistently across duplicated code. Of course, the best scenario is to remove duplicates altogether, if possible.

To answer the question whether code duplication may create problems during refactoring, we computed both internal and external code duplication using the clone detector from [Wettel and Marinescu 2005]. This clone detector was preferred as it is very easy to configure, works at a lexical-syntax level (as opposed to more complex clone detectors that need semantic analysis), is robust, and freely available. For the clone detector, and a minimal duplication block size of 20 lines - that is, similar blocks of smaller size are not considered to be duplicated, but coincidentally similar. Figure 9 shows the results. The top image shows those files from the file-folder-module hierarchy which contain external duplication. The arcs connect files sharing duplicated blocks. The results look good. First and foremost, there is little *external* duplication. Secondly, duplicated blocks are shared among a few (2..3) files, not more, *i.e.* the same block is not found to be duplicated across many files. Thus, removing them altogether should not be very hard. Finally, about half of the files sharing duplicated blocks are located in the same modules - this is visible as the short curved edges in Figure 9 top. All in all, the conclusion drawn was that external duplication is not a serious problem for system-wide refactoring or testing.

The bottom image in Figure 9 shows internal duplication. The color map indicates the number of duplicated blocks per module, red being all modules with 60 or more duplicated blocks, and green being modules with a single duplicated block. Gray indicates modules with no duplication. This image is less positive than for external duplication: about half of the modules have at least one duplicated block, and around 10 of them have duplication levels above 30 blocks per module. Also, notice the high correlation between modules containing duplicated blocks and modules having many lines of text or code, and a high complexity (Fig. 6).

4.5 Documentation analysis

After the duplication analysis, the consensus strengthened that refactoring would be doable. However, given the architectural decay and team and communication problems mentioned before, the question appeared: how easy would it be to actually understand what is in the code to refactor? We performed a documentation analysis to shed some light on the state of documentation. For this, we used the visual evolution analysis tool and methods described in [Voinea and Telea 2007; Voinea and Telea 2006]. After loading all file versions in the tool, we first clustered files by type in two groups: documents (`.doc`, `.html`) and all other files. Figure 10 top shows these two groups. Within each group, files are

bottom-up in creation order (that is, older files at the bottom). Red dots indicate file changes. We see now that documents represent over a third of all files in the project (which is a quite high proportion for a code base).The number of documents increases in sync with the number of other files, but with a slight delay (see the figure), which is expectable. A very good sign is that document files appear to be well maintained - the red dots indicating change are relatively as frequent in the document set as in the non-document file set.

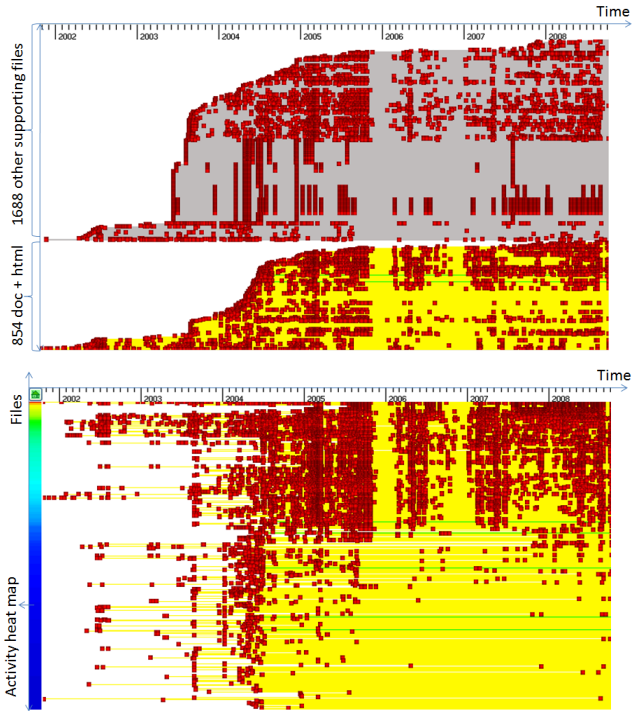

Figure 10: Documentation evolution. Top image: changes in documents (yellow) vs other files (gray) sorted by time. Bottom: documents sorted top-down by activity

The bottom image in Fig. 10 shows a zoom-in on the document files only. The files are sorted in this view in inverse order of activity, that is, files at the top changed the most while files at the bottom changed the least. This is also visible in the density of the red dots. This image seems to be split in two parts: the top part of about 40% of all files, *i.e.* the area in the image having a very high concentration of red dots, shows a set of documents which are all modified very often, over the entire project duration. The rest 60% files, *i.e.* the area being mainly yellow and having few red dots, is modified very sporadically, except during the early period of documentation creation. This suggests two sets of documents: and up-to-date one which can be used immediately during refactoring, and a passive one, which is not maintained, and which should better not be used during refactoring as it does not contain updated information. We explain this as follows: Since virtually all source code files are heavily modified throughout the entire project (see Fig. 2), there is a high chance that their documentation (which is not modified for many of them, as seen in Fig. 10) has gotten out of sync. Thus, care should be taken when considering this unmaintained documentation during refactoring.

5 Discussion

Below we discuss the most important insights gathered during our study.

5.1 Findings

Recapitulating from Section 4, the main conclusions drawn by the stakeholders, based on our analysis, were as follows:

- the project's evolution had two main phases. Phase 1 (first two years roughly) showed sustained development, the creation of over 75% of the total code and documentation, and a relatively good overall progress. Phase 2 (last four years) showed mainly work on closing old modification requests, little introduction of new features, increased lagging behind the deadlines, and a deterioration of the initial 'clean' architecture and design.
- a main cause of the suboptimal progress was the assignment of critical requests and most complex subsystems to a team which had communication issues and apparently also less performant skills than the other two teams.
- all studied indicators show a stagnation of the development in the last years; activity mainly tries to fix problems from the past
- the architecture, code quality, and documentation level of the final product are not ideal, but it is clear where the structural problems are. Refactoring and code quality improvement is possible with limited effort.

Overall, the stakeholders decided that the current software product is close to the desired end target. To complete the product within time, refactoring and code cleanup steps would be taken to remove the detected problems; and a different team assignment would be done to focus the highest-quality workforce on the most critical parts. All in all, the conclusion was that the root cause of the suboptimal evolution was a *process* problem (team assignment), which created the *product* problems (code and architecture).

5.2 Methodology

Given strong time and cost pressure (decisions had to be taken quickly and with little investment), we could not approach this assessment by doing an in-depth analysis of the process and product. Moreover, the quite specific nature of the developed software would have made such an assessment hard for most outsiders. Hence, we chose a different path. We extracted all static and evolution information from the repository and contained code, and prepared a number of visualizations, as the ones shown in the previous section. Next, we presented these visualizations during an interactive workshop involving both technical and non-technical stakeholders (designers, architects, and managers).

We specifically refrained from drawing our *own* conclusions from these images, and limited our comments on the visualizations to explaining what kind of data is shown, and how to interpret the drawings. At that point, the stakeholders started having specific comments and questions on the shown images, for example the request to zoom in specific parts or show other attributes or relations in a view. Given that a) all static data were extracted upfront; and b) the involved visualization tools allow quick, interactive changing of the attributes shown and other visual parameters, it was quite easy to answer such requests in a matter of minutes.

Once we noticed that the stakeholders formed some clear hypotheses or early conclusions, we tried to generate new views (with other attributes, metrics, or visual correlations) to (in)validate these points. This iterative process was quite fast. The result discussion took around 5 hours, out of which about two were actually spent after the conclusions were drawn, on determining which is the best way to continue.

We believe this way of working to be more efficient and effective than other possibilities. In the literature, we mainly encounter two

other assessment methodologies, as follows:

- giving the analysis and visualization tools to the customers (classical tool vendor scenario)
- assessing the product/process offline, and showing the conclusions to the customers (classical consultant scenario)

We see important risks in both above cases. Giving the tools to the customers may look attractive as it theoretically permits them to create any investigation scenarios they want (in the limit of the tool support, of course). However, this also poses a high burden on the time investment of the stakeholders; and requires them to be highly familiar with the tools and all their options and limitations. This solution works well if there is an in-house tool expert in the stakeholder organization. If not, tools tend often to be mis-used or even not used at all after initial adoption (we have encountered numerous such cases). Assessing the data offline by a consultant and presenting conclusions is also risky, as it implies that such conclusions can be reliably drawn from the available data. More often than not, a project's context is very complex and involves a lot of factors which are hard, if not impossible, to extract purely from a process/product analysis. Presenting a wrong conclusion to a customer is highly risky, as it will decrease the customer's confidence in the analysis method, tools, or third-party (consultant).

Apart from these points, we noticed an additional factor that favors the methodology used by us. During decision-making or post-mortem project analyses in organizations, often different parties are involved, *e.g.* management and technical people in our case. Such parties may have different, conflicting interpretations of existing data or past events. Having an *external* party provide the analysis (but not draw the conclusions), like we did, increases the chances of acceptance within the organization, and decreases the time and cost of the assessment. Using a *visual* analysis increases the communication bandwidth between technical and non-technical stakeholders. Effectively, this way of working empowers the stakeholders to have the data they need to draw conclusions, but does not force the conclusions upon them.

5.3 Simplicity

Although this may be a controversial point, we strongly believe from this case study and other similar ones we have done in the past (not presented here for space limitations but sharing the same tool combination and tool usage pattern), that *simplicity* is the key to making visualization gain wider acceptance in software engineering at large. Visualization is not a core technique in the field, like *e.g.* compilers, profilers, debuggers or other tools of the trade, but rather an 'underdog', if we can use this expression. As such, it has to provide clear benefits with limited investment. To achieve this, it has to be easy and fast to use. This involves, among others, a perfect and quick integration with the overall tooling environment; the possibility to generate comprehensive overview images in virtually no time; and the ease to pose queries and execute drill-down actions on such images. The types of views we used here, *i.e.* treemaps, dense Cartesian 2D pixel layouts, and hierarchically bundled edges, all meet these requirements. In particular, the presets of all these visualizations work well - it is only when drilling down or selecting a subset of the data that the user has to do any parameter setting.

5.4 Feedback from customers

It is insightful to detail the customer context and feedback, as follows. The persons involved in the analysis can be split into two types: management and technical. It is important to note that *most* of the low-level findings, *e.g.* individual dependencies between specific files, the identity of developers working on specific project components, changes done to the documentation, and even the identity of the most complex, or largest, components in the system were quite well known before we showed the images.

However, the project owners stated unanimously that they found this type of analysis, and way of working, useful. In this case, the question may come: What was the added value of our analysis? The customers mentioned the following points as being the major added values:

- a combination of process data (teams, modification requests, evolution data) and product data (dependencies, code duplication, static code metrics) is very useful as it addresses more involved questions than just one data type at a time;
- seeing as much data as possible on a single screen, but also allowing drill-down, is good as it supports the 'plenary meeting' type of discussion but also allows specific questions to be answered directly during such meetings;
- having the stakeholders draw the conclusions, rather than having them served by the external analyzing party, is absolutely essential;
- clear (and low) cost-and-time limits on the entire assessment are vital for getting them accepted by higher-level management.

Also, the stakeholders stated that using such types of analyses continuously, and from the beginning of, new projects would be of clearly added value in early detection and discussion of problems. To do this, all above points have to be met from the very beginning, otherwise the apparent added value of visualization may exceed its benefits.

We also noted that the visualizations, and result presentation method used, was extremely instrumental in helping the two types of customers (management and technical) *communicate* and reach an *agreement*. After our assessment and presentation, it became to us apparent that some of the insights were actually known to part of the team, but not universally accepted by the others. As such, the visualizations were instrumental in showing objective evidence and helping all stakeholders reach a consensus. Of added value was, again, keeping the visualizations *simple*, so they could be understood with little or no explanations by both management and technical persons.

As limitations of our (visual) analytics method, we note the following points. First and foremost, questions such as "what to do from now on to solve our problems" are well-known to any consultant. Visualizations can only give support, but not answer such questions directly. Secondly, "what if" scenarios need a tighter combination of interactive analysis and presentation. For example, users would like to know what would happen if a given refactoring were done. Performing such what-if scenarios in near-real time is however extremely challenging, and it requires novel ways to empower users to specify code or design modifications quickly (ideally, on the images themselves), execute those on the code, redo the static analysis, and redraw the images. Last but not least, management mostly thinks in terms of *value* and *waste*, not lines of code, dependencies, and changes. Although value and waste are well-known terms to the newly emerging agile and lean development communities, there are still no automated ways (models and tools) that allow mining and visual presentation of such metrics in correlation with, and based on, solely data stored in software repositories. Designing such tooling support would be of a huge value in making visual analysis tools more accepted, and actually more useful, in software-intensive organizations.

5.5 Threats to validity

There are several threats to the validity of the presented methodology. Here, we focus specifically on the threats to the proposed analysis *method* in general, *i.e.* not specific to the concrete project on

which we applied it. This type of insight should be more valuable to the re-application of the proposed methodology in other projects.

Concerning the validity of the *data*, there is a non-negligible possibility that the modification requests (MRs) are not fully reliable in a software project in general. In our specific case here, however, the team stated that the MRs (date, affected file, person filing them etc) were carefully maintained, as this is a company policy. Although this cannot be independently verified, the sheer amount of MRs stored in the configuration management system, and a manual examination of several MRs chosen by us at random, strongly suggests that a disciplined approach was used here and that this data is of good quality. Moreover, note that our analysis is not sensitive to small-scale outliers *e.g.* a few incorrectly filed or missing MRs. What is essential is noticing strong trends defined by cumulating individual MRs.

Concerning the preciseness of the *analysis* (metrics, C/C++ code analysis, dependency extraction), the analyzer we have used, which is of our own construction, was extensively tested on very large software projects containing complex code [Telea and Voinea 2008a], as well as on the gcc testsuite. We are confident that the delivered results are correct - incorrect results, if present, would have been signaled as parse or type checking errors.

6 Conclusions

In this paper, we presented the process of using a combination of static and evolution analysis and visualization tools for supporting decision-making in the framework of large-scale industrial software projects. Our aim has been to demonstrate how the utilization of existing static analysis and visualization tools and methods can be highly cost-and-time effective in helping decision making in the management of software projects. The case presented here is typical in several respects, including the involved data (typical industrial C source code stored in a mainstream source control management system), the process (multisite development of a mid-size embedded software product over several years), and the goals (plan the future project course based on data mined from the past, with a minimal time and cost investment). Correlating evolution and static analysis data, presenting this via scalable, dense-pixel visualizations, but letting the actual stakeholders draw the conclusions, seems to be a very effective way for software visualization to bring measurable contributions in a budget-tight industrial context.

We are currently testing this methodology on several other industrial projects, with preliminary good results. Furthermore, we want to study the possibility of simplifying the assessment process by assembling even higher-level visualizations from existing components, thereby targeting a larger spectrum of concrete analyses.

References

ABELLO, J., AND VAN HAM, F. 2004. MatrixZoom: A visual interface to semi-external graphs. In *Proc. IEEE InfoVis*, IEEE Press, 183–190.

BALZER, M., AND DEUSSEN, O. 2005. Voronoi treemaps for the visualization of software metrics. In *Proc. ACM Softvis*, 165–172.

DIEHL, S. 2007. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer.

HOLTEN, D. 2006. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG 12*, 5, 741–748.

KEIL, INC. 2008. The Keil C166 compiler reference manual. `http://www.keil.com`.

KOSCHKE, R. 2003. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance: Research and Practice 15*, 2, 87–109.

LORENSEN, B. 2004. On the death of visualization: Can it survive without customers? In *Proc. NIH/NSF Fall 2004 Workshop on Visualization Research Challenges*, NIH/NSF Press.

REISS, S. P. 2005. The paradox of software visualization. In *Proc. VISSOFT*, IEEE Press, 59–63.

SPENCE, R. 2006. *Information Visualization*. ACM. Press.

SPINELLIS, D. 2008. The CScout C extractor. `http://www.spinellis.gr`.

TELEA, A., AND VOINEA, L. 2008. An interactive reverse engineering environment for large-scale C++ code. In *Proc. ACM SoftVis*, 67–76.

TELEA, A., AND VOINEA, L. 2008. SOLIDFX: An interactive reverse-engineering environment for C++. In *Proc. CSMR*, 320–322.

TELEA, A. 2006. Combining enhanced table lens and treemap techniques for the visualization of large data tables. In *Proc. EuroVis*, IEEE Press, 13–20.

VOINEA, L., AND TELEA, A. 2006. Multiscale and multivariate visualizations of software evolution. In *Proc. SoftVis*, ACM, 115–124.

VOINEA, L., AND TELEA, A. 2007. Visual assessment of software evolution. *Science of Computer Programming 65*, 3, 222–248.

WARE, C. 2004. *Information Visualization, Second Edition: Perception for Design*. Elsevier.

WETTEL, R., AND MARINESCU, R. 2005. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Proc. Intl. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, IEEE Press, 201–209.

WONG, P. C., AND THOMAS, J. 2004. Visual analytics. *Computer Graphics and Applications 24*, 5, 20–21.