A Visual Analysis Approach to Support Perfective Software Maintenance

Jonas Trümper Martin Beck Jürgen Döllner
Hasso-Plattner-Institute - University of Potsdam, Germany
{jonas.truemper|martin.beck|juergen.doellner}@hpi.uni-potsdam.de

***Abstract*—Ensuring code quality represents an essential task in «perfective software maintenance», which aims at keeping future maintenance costs low by facilitating adaptations of complex software systems. For this purpose, changes and related efforts have to be identified that imply high positive impact on future maintainability. In this paper, we propose a novel assessment method that applies visual analysis techniques to combine multiple indicators for low maintainability, including code complexity and entanglement with other parts of the system, and recent changes applied to the code. The approach generally helps to identify modules that impose a high risk by causing increased future maintenance efforts. Consequently, it allows for exploration, analysis, and planning of different preventive measures that, e.g., most likely will have a high return on investment. In our tool implementation, we use circular bundle views, extended by the third dimension in which indicators can be mapped to. We have evaluated our approach by conducting a case study based on our tool for a large-scale software system of an industry partner.**

***Keywords*-Software maintenance; Visualization; Quality management**

I. Introduction

Adapting software systems to changing requirements is inevitable to keep them useful after their initial release [7, 21]. Repeated adaptation, however, typically leads to significantly reduced maintainability, commonly termed *decay* [12]. *Perfective* and *preventive* software maintenance [1] both act as established countermeasures, i.e., help to keep software systems *maintainable*. They include a particularly important objective, namely ensuring code quality. Nevertheless, responsible personnel often faces dilemmas, such as: Where and how to invest the limited resources for an optimal positive effect on future maintainability? The given resources typically do not suffice to fix all quality issues in the code. In addition, the most prominent quality issue may well be the hardest to fix. So responsible personnel will carefully consider the pros and cons of any action that refers to code quality before it is undertaken. They will only consider approving such action if its chance to increase or ensure future maintainability outweighs its costs. Yet, weighing these pros and cons in practice is typically based on subjective judgment obtained by experience, discussions with developers, etc. Interestingly, there is generally a lack of specialized visualization techniques and tools.

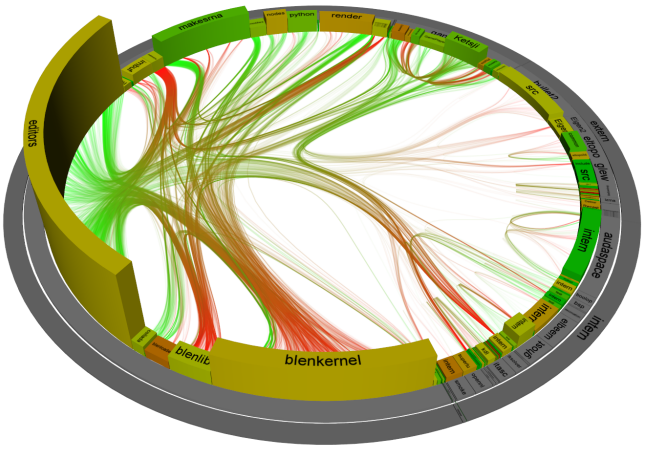


Figure 1. Example visualization using a three-dimensional circular bundle view. System structure is mapped to the outer rings; dependencies are mapped to edges within the circle. Metrics are mapped to color, height and width of ring elements.

Assuming any actions can be identified, approval by other stakeholders, namely management, is still difficult to obtain. Management tends to favor other, visible changes, e.g., to external software quality such as new features or fixed bugs [5]. This is due to a number of reasons, including 'invisibility' of internal software quality to non-development staff and non-measurable return on investment for effort devoted to it. Consequently, it is important not only to find reasons why a number of actions should be undertaken first, but these reasons also have to be visible to and reasonable for all stakeholders.

We propose a novel decision-supporting technique that uses enhanced circular bundle views [18] to help identify modules[1] that impose a high risk by causing increased future maintenance efforts. It further enables exploration, analysis and planning of preventive measures to modules with maintainability problems that should be tackled first. Insofar, our method provides clearly understandable *indications* within a single view. In particular, it serves to bridge

[1] *Module* describes a piece of code that implements specific functionality, e.g., a class.

the communication gap between technical and non-technical staff in software projects.

The visualization technique combines multiple indicators such as current code complexity, past change frequency, and entanglement (Fig. 1). In contrast to existing techniques, integration of entanglement analysis facilitates estimating to what degree other parts of a system rely on a module's correct behavior. Moreover, combining the indicators in a single view a) facilitates comparison of the indicators in a stable context (the system structure) and b) supports identifying phenomena that are related to not only modules, but to both modules *and* dependencies, such as low maintainability of a module and many dependencies to this module. We have implemented the technique as a tool that uses circular bundle views, extended by the third dimension, which adds a visual dimension that indicators can be mapped to. Further, we evaluate our approach and tool by a case study conducted on a large-scale, grown software system.

II. Related Work

Numerous approaches exist to estimate current maintainability [26], future maintainability [23], future maintenance activities [17], or even maintenance effort as person hours or costs [15]. While they are typically based on statistical methods such as principal component analysis or multiple regression, some also use neural networks and fuzzy logic. These approaches mostly do not capture entanglement, which significantly influences maintainability, i.e., the effort required for changes, and impact of changes. Further, they compute estimates that refer to an entire system and thus are too coarse-grained to identify potentials for improvements in maintainability.

Visual approaches can be used as well as vehicle for communication between technical (e.g., development) and non-technical staff such as management. In addition, since they typically do not compute a final result, expressed for example by a single number, such as "class complexity", they allow for more detailed analysis on demand.

Baker and Eick [3] depict evolution of software metrics as charts, visually embedded into a treemap's nodes. Additional animation allows for comparing different snapshots of a system's evolution. Balzer et al. [4] propose a Voronoi-based treemap visualization that improves visibility of hierarchical relationships, while our three-dimensional approach features additional visual variables and supports entanglement analysis. Langelier et al. [19] present three-dimensional treemaps and sunbursts with metric values projected to software artifacts' visual representations. In addition, they visualize evolution by depicting evolutionary steps next to each other. Wettel and Lanza [31] also use treemaps to depict software systems as virtual cities, although they do not analyze a system's evolutionary steps. Bohnet and Döllner [5] extend these approaches by an additional timeline depicting change frequency that allows for restricting the analyzed time interval. We use a different visualization concept that additionally supports entanglement as indicator.

Lanza and Ducasse [20] present "polymetric views" that represent node-link diagrams with metrics mapped to visual variables of nodes. In contrast, their technique does not scale well for large graphs and typically suffers from visual clutter caused by crossing edges. Eick et al. [11] use multiple visualization techniques, such as (matrix) plots, bar charts, and graphs, to depict metrics, code evolution, and dependencies between change requests. The graphs used, however, ignore essential hierarchical relationships of analyzed software systems. Alam and Dugerdil [2] extend treemaps by relationships that connect nodes with bended edges from above. Yet, edges are not bundled, resulting in visual clutter by many edge crossings and occlusion of treemap structures. Gall et al. [14] use a visualization similar to cone trees [24] to depict system structure and bar representations to encode attribute percentages per module. In contrast, their visualization does not support entanglement analysis.

Lewerentz et al. [22] use clustered graphs with force-directed layouts. They map metrics to visual variables of nodes and to forces used to layout the graphs. The graphs, however, quickly become cluttered by crossing edges and nodes can occlude edges. Telea and Voinea [27] analyze the inability of a software project to keep its planned time frame in a case study. For it, they use circular bundle views to identify architecture violations, but do not combine these views with metrics or change history.

CppDepend[2] is an industrial tool that enables visual analysis of a software system's metrics and dependencies. However, analysis of evolution history is not supported, and its dependency graph does not scale. Bourquin and Keller [6] use Sotograph[3] to manually identify architecture violations as base for refactoring opportunities that have a high-impact in terms of improved code quality. They ignore code change data and, thereby, risk 'generating' unnecessary maintenance effort on code that would otherwise not be part of (future) maintenance activities. Wilhelm and Diehl [32] compute layered layouts for node-link diagrams based on design quality metrics such as abstractness to check for forbidden edges that cross layers in the wrong direction. The computed diagrams, in contrast, typically suffer from crossing edges, especially with large input data.

III. Visual Analysis by Combining Multiple Software Quality Indicators

Within the software maintenance cycle, regular countermeasures against software decay are typically applied (Fig. 2): (S1) Assessing maintainability problems, (S2) determining a ranking of these problems, (S3) picking some

[2]http://www.cppdepend.com/, as of 01/26/2012

[3]http://www.hello2morrow.com/products/sotograph/, as of 01/27/2012

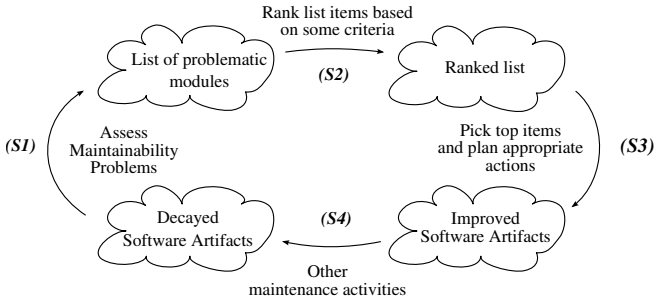


Figure 2. Typical software maintenance cycle: Steps S1, S2 are (both adressed by our technique) and S3 act as countermeasures against (accidental) software decay (S4).

topmost items of that ranked list, and (S4) addressing them. Our proposed method addresses steps 1 and 2 by combining multiple indicators for low maintainability in enhanced circular bundle views (CBVs). Among other indicators, it includes entanglement with other modules, which is a more complex measure and traditional methods (such as metrics) do not support its analysis.

A. Proper Entanglement Analysis is Crucial

Software metrics typically measure quality indicators, such as encapsulation, at class or method level by computing a single value. If entanglement is measured in this manner, e.g., as fan-in, fan-out [13], or coupling between objects [9] coefficient, a number of individual incoming or outgoing dependencies of a module are reduced to a single value. From the perspective of software architecture, some of these dependencies are allowed, while others are not. Violating dependencies can also have different severity, i.e., one forbidden dependency may be worse than another. A single metric value does not properly reflect this since it lacks precision and sufficient detail, i.e., provides only an abstract or aggregated result. By detailed entanglement analysis, in contrast, we are able to estimate to what degree other parts of a system rely on a module's correct behavior. That is, the more entangled a module is, the more likely it is that changes to its behavior break relying modules.

B. Context Dependency

For assessing internal software quality, analyzing a single indicator (e.g., a complexity metric) is not sufficient [5]. More precisely, the 10 most complex code modules of a software system do not necessarily represent the 10 most important quality issues if they comprise only a few lines of code. By contrast, if the 10 medium complex modules have significantly more lines of code than the average module, they are probably harder to understand. Consequently, a combination of multiple software metrics allows for more precise assessment of a system's actual code-quality problems. Even more, if a module is considered complex according to multiple software metrics, it may be only a small factor for maintenance: Less complex modules that were recently touched (e.g., during the last months for every third feature extension), and are weakly covered by tests are probably a bigger quality problem that should be tackled upfront. In other words, recent change frequency is a good predictor for code maturity [8]: The more recent changes affect a module, the less stable its implementation is.

Still, proper interpretation of most software metrics is typically a matter of *context*. For instance, there is no universal threshold that one could apply to determine whether a module has *low* or *high* code complexity [5]. A module that implements a complex algorithm is probably allowed to have a relatively high complexity value while not being considered *too complex* at the same time. Other modules with lower complexity value, however, might be considered to exhibit high complexity. Another example for context dependency is that interpretation of a metric's value for a single module is dependent on *all* other modules within the project. So, for identifying outliers it is necessary to relate each value in a set of values to all other values in this set.

C. Visual Analysis: Extending Circular Bundle Views

Our approach provides visual analysis for supporting perfective maintenance by implementing an extended CBV [18]. CBV is a two-dimensional information-visualization technique for layouting compound-directed graphs, i.e., a hierarchical graph annotated with directed relations between its nodes. CBVs are similar to icicle plots, except that they use concentric rings instead of a rectangular area to encode hierarchical containment of nodes. Node attributes are typically mapped to the visual variables size and color of ring elements. Non-hierarchical relationships between nodes are then depicted in the free inner circular area and relationship attributes are mostly mapped to their visual variable color. As graph, we use a system's hierarchical module structure and dependencies between modules as relations.

We add two main extensions to the original CBV: First, we extend its data model by additional attributes to each node. They correspond to user-configurable software metrics or developer activities, which are aggregated to parent nodes. Although the type of aggregation is user-configurable, software metrics have to be chosen with care because metrics aggregation is subject to research and might be misleading to users [29].

Second, we extend the original CBV by the third dimension (Fig. 3) to add an important visual dimension: *height*. Being able to integrate more visual variables in a single view than with the original CBV, less context switches to other views are required for users to assess the same visualized data. We further chose height instead of other, possible visual variables (e.g., texture, color saturation or luminance) since height can be better visually distinguished from already

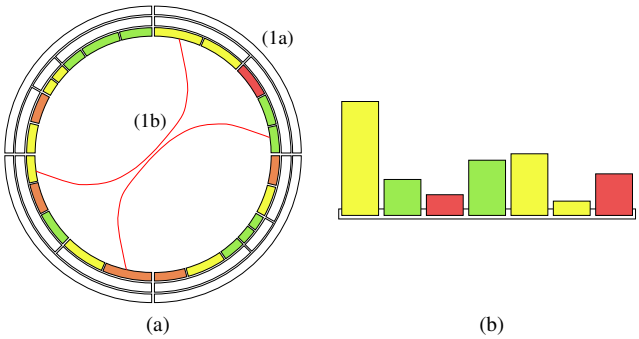


Figure 3. Conceptual sketch: a) Top view onto a 'traditional' circular bundle view with hierarchical relationships projected to the outer circles (1a) and non-hierarchical ones in the inner circle (1b). b) Side view, metric values projected to node height.

present variables. In addition, variables such as texture may interfere with text labels, rendering the latter unreadable.

Humans generally compare three-dimensional objects – if there is no particular context – by their volumes instead of by their three separate dimensions [28]. Here, however, the context defines that the dimensions width and height of a 3-dimensional ring element have a distinct meaning and can thus be taken into account separately when comparing two objects. Adding height, however, introduces an occlusion problem: High modules may occlude child modules or even dependency edges. We address this by (1) only mapping values to a module's height on the innermost ring because we argue that those modules are more important than modules on the outer rings and (2) making modules transparent when users hover those modules or other modules connected by a dependency edge.

To be able to analyze different hierarchy levels concurrently and to compare attribute values across these different levels, we normalize attribute values of all *visible* modules on the innermost ring. The lowest value is mapped to zero and the highest value to one. Consequently, the mapping is recomputed each time the set of visible modules changes.

The ExtraVis CBV implementation [10] added collapsing of modules to aggregate its submodules and dependencies to and from them. However, it routes all incoming and outgoing dependencies to the center of a collapsed module, which hinders assessing the distribution of dependencies to hidden submodules. To alleviate this, we fan out all dependencies across the full angular size of collapsed modules.

We further complement the CBV technique with a drill-down approach instead of full-detail views as in the original CBV. We argue that users are more interested in gaining an overview of a software system's structure in the first place. Three drill-down operations are available: (1) *Collapse/expand single subtree* enables users to selectively drill down into a subtree of the overall hierarchy. (2) *Collapse/expand per depth level* allows for step-wise analysis per level of a hierarchy. Descending one level shows deeper

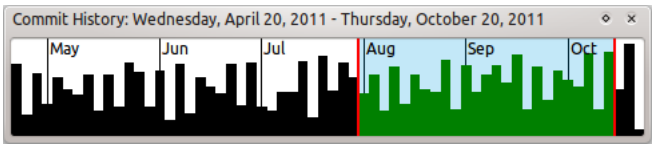


Figure 4. Timeline plot of development activity. Selected timerange (08/2011 to 10/2011) is highlighted.

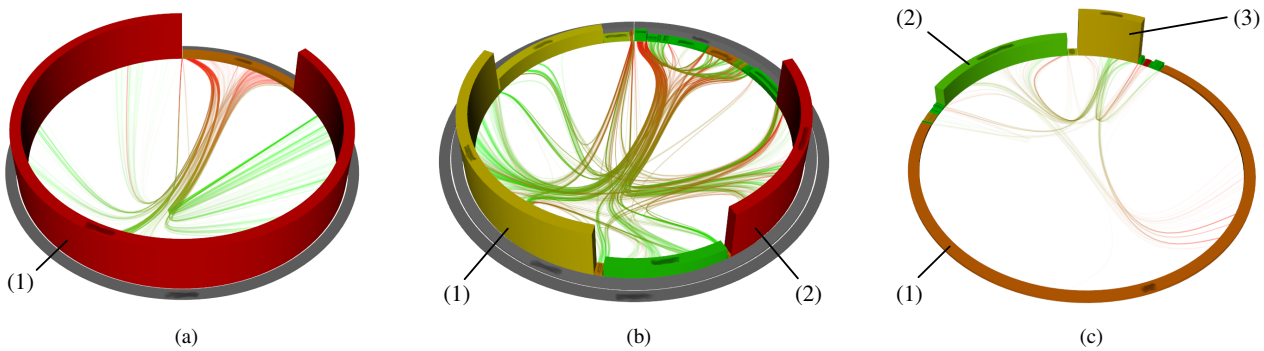
nested modules while still retaining an overview of present dependencies. (3) *Restrict to subtree* takes a module as input and reduces the depicted hierarchy to children of this module for detailed analysis. While reducing visual clutter and helping to focus on details of the subtree, it also hides dependencies to and from modules outside of the selected subtree.

An additional timeline plot shows development activity (Fig. 4): The bar chart plots the number of commits for each day of development. Using the plot, users can select a time range and thereby filter displayed data by this range.

D. Benefits of Our Approach

Depicting the whole system structure and all modules' measured attribute values in three dimensions, and dependencies simultaneously, provides the following benefits:

1) Communicating analysis results to all participating stakeholders is facilitated. With outliers in all measured indicator dimensions becoming visible and recognizable, developers, software architects, and managers get means to communicate about internal software quality that can be discussed and interpreted by all parties. In addition, analysis results are embedded into the architectural context of analyzed systems.
2) Visual analysis is lightweight, fast and uses quasi-standard indicators. Management can use our technique as an instrument for determining the success of actions for improving internal software quality on a regular basis.
3) Our approach enables a human's visual system to fully exploit one of its strengths, namely identifying outliers using pre-attentive perception [16, 30]. Outliers in metric values are easily distinguishable by notably different height or color. Due to the CBV's edge bundling, outliers in dependencies visually differ from other dependencies by not being bundled.
4) Ad-hoc interconnection of multiple indicators that does not require a defined mathematical operation for interconnection becomes possible by mapping the indicator's values to visual variables.
5) By filtering indicators by time range, their progression over time can be analyzed. By this, trends can be spotted and preventive actions can be taken before outliers become a serious problem.



(a) (b) (c)

Figure 5. Drilling down into the module hierarchy of Francotyp Postalia's controlling software. We descend in the hierarchy (a and b) and restrict the view to a subtree in c).

IV. Case Study

We have performed a case study for an industrially developed software system using a prototypical implementation of our tool. Francotyp Postalia GmbH[4] is one of the world's largest vendors for franking machines. We analyzed an embedded software system for one of their machines. This software system is written in C/C++ and consists of approximately 900,000 SLOC (source lines of code) with an average of 10 developers working on it for more than two years. While most of its development takes place internally, some modules and libraries were purchased from third party vendors.

A. Setup and Configuration

Our tool extracts an attributed compound-directed graph from a C/C++ software system by retrieving its file system hierarchy, static dependencies, and file-related software metrics. While dependencies are directly mapped to the CBV's edges, we attribute the graph with metrics per file and map them ($\mapsto$) to visual variables as follows:

- *Code complexity* $\mapsto$ *module color (green to yellow to red):* is computed as ratio of number of statements in nesting level three and deeper, *NL3+* [if(...) { if(...) { while(...) { } } }], to source lines of code (SLOC). Deeply nested code is commonly considered to be hard to understand and, thus, a high percentage of deeply nested code implies high complexity.
- *Relation direction* $\mapsto$ *edge color (green: start, red: end)*: Similar to the color-coding of modules, potentially dangerous relations of a module (i.e., incoming - other modules rely on that module's implementation) visually stand out.
- *Change frequency* $\mapsto$ *module height:* includes any modifications applied to a file within the last 6 months according to a project's source-code management (SCM) system.
- *Size* $\mapsto$ *angular size of module:* is computed as SLOC.

[4]http://www.francotyp.com/, as of 01/25/2012

Thereby, we can easily identify large, complex modules that have recently been touched by developers. By sizing up incoming and outgoing dependencies, we can assess a module's entanglement with the rest of the system.

In our use case, we adopted a common color scheme of the domain, i.e., the traffic-light metaphor. Nevertheless, the scheme can be modified as necessary, e.g., to bypass deficiencies in human color perception [25].

B. Tool Use in Practice

During Francotyp Postalia's review sprints, their development teams typically perform quality-improving tasks such as refactorings or reengineerings. Developers collect these tasks over time during their daily work and, at the beginning of each sprint, managers, project leaders, and developers prioritize them together. However, this task list is not necessarily comprehensive because developers tend to focus on code-centric improvements. By contrast, project leaders and managers want to concentrate on modules that are relevant for future maintainability but they lack detailed system knowledge. Thus, misunderstandings about relevance and priority of a task are common.

We have applied the tool to improve this situation and support Francotyp Postalia's managers and project leaders with understanding potential maintainability risks in their software systems. Fig. 5 illustrates an exemplary analysis process for their franking machine's software system using our extended CBVs: We start by analyzing the software system at its uppermost hierarchy level. At the first level, (Fig. 5a), we can distinguish two modules: A large, complex module that contains product-specific code (1 in Fig. 5a) and a smaller, medium complex module that represents Francotyp Postalia's product-independent libraries. While the former has been modified frequently, the library module has barely been touched, indicating mature code. However, comparing code sizes, we can see that the base library is very small as opposed to a large product-specific code base. There might be potential for increasing code reuse between different franking machines.

Descending another level within the software system's modular structure (Fig. 5b) reveals that the product-specific code is composed of multiple sub-projects, which depend on each other. However, predominant projects from a manager's point of view such as country-specific application projects are hardly visible here because they consist of only a few lines of configuration code. Instead, visual focus lies on two large support projects (1 and 2 in Fig. 5b) that have been heavily modified and expose high complexity in relation to others.

To further analyze these modules, we investigate the more complex one first (Fig. 5c): We apply the "restrict to subtree" operation to this module, thereby hiding all other modules but the selected module and its submodules. Disturbing information about currently irrelevant modules is hidden and leaves additional space for submodules on the inner rings. Here, we can see a large module with mid complexity (1). However, it exhibits a relatively low change frequency. This module is a third-party library that takes care of display devices and manages the graphical user interface. Despite its high complexity and strong entanglement with the system, this module should not be a major subject to quality-improving measures because it is modified infrequently and, thus, seems to be in good shape.

Instead, there are two other modules visible that expose high change frequencies, medium and low complexity, and that are relatively large (2 and 3 in Fig. 5c). Both of them deal with the franking machine's user interface and control the franking machine's human interface. Bugs at this level will definitely affect customer satisfaction. Thus, they should be subject to perfective maintenance.

To check these two modules' entanglement with other modules, we switch back to an overview of the full software system with a hierarchy depth of three. We notice a strong entanglement of both modules (1 and 2 in Fig. 6), which indicates that other parts of the system are relying on them. Yet, this overview reveals other relevant modules: First, the largest module from Fig. 5b is separated into its submodules. An outlier among them exposes high complexity and medium change frequency (3).

Second, we can easily distinguish the two most complex modules within this view because their red color contrasts with surrounding green modules (4 in Fig. 6). Furthermore, they also have a strong system entanglement. However, their size in relation to other modules on this hierarchy level shown is rather small and they have almost no modifications during the last six months. Nevertheless, they are both submodules of a printing module within Francotyp Postalia's product-independent library, which renders them crucial for franking machines. Although the probability of future maintenance work within these modules seems to be low, we suggest further investigating why these libraries have such high complexity.

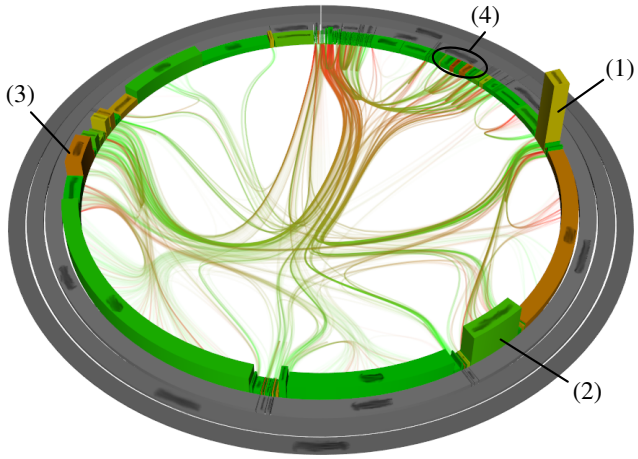


Figure 6. An overview of Francotyp Postalia's software system at hierarchy level three enables to identify outliers in change frequency and complexity.

During our discussions with the development teams, project leaders, and managers, our images and interactive visualization tools proved as a viable vehicle for communication. By hiding irrelevant information such as very small modules we could focus on outliers with high complexity, strong entanglement with the system, and frequent modifications. Furthermore, in the extended CBV, our users had no problems comparing the distinct variable dimensions width and height of the three-dimensional ring elements separately. We still noticed a few wishes for enhancements that targeted supporting widgets and analysis functionality. Among others, theses included highlighting forbidden dependencies that violate the intended system architecture (which first requires a white- or blacklist for dependencies), more flexible undo/redo functionality, and support for storing and restoring view configurations across sessions.

Overall, visually combining multiple indicators, which are easy to explain and understand, helped the team gain new insights and encouraged further discussions: "emphasizes problems not recognized before". In the meantime, we improved the implementation, and the tool is now actively used.

C. Threats to Validity

Our case studies' validity is prone to several threats. First of all, the studies can hardly provide a proof for maintenance costs reduction. Yet, we argue that we showed multiple benefits of our approach by selecting a single but reasonable complexity metric (nesting level), demonstrating our analysis process, and by providing multiple indicators to a few places within the source code. Although our selected metrics, i.e., SLOC for module size, NL3+/SLOC for complexity, and recent modifications for developer activity, are arguable, our concept is not limited to these specific metrics.

V. Limitations

While conducting the case studies, we identified a number of limitations of our approach; however, most of them can be at least alleviated. First, the human visual system has difficulties in discerning different color intensities, especially when the visual difference is small [30]. Hence, modules with attribute values that are numerically close to each other, but not equal, may nevertheless be seen as having equal values by humans. However, our tool permits to freely configure the color mapping so that it can be adjusted to discern numerically close values as well. Moreover, users cannot configure if attributes mapped to angular size shall be aggregated and how and aggregation of angular size automatically propagates up to higher levels in the depicted hierarchy. This limitation, however, is inherent to the visualization concept.

Since attribute values are first normalized before being mapped to visual variables, absolute values cannot be 'read' from the visual representation. This limitation can be mitigated, if not solved, by tooltips showing absolute values, which are displayed while hovering over modules. Yet, they were rarely used during our case studies. Additionally, dependencies that are invisible at compile time, such as those introduced by late binding, cannot be analyzed.

Conclusions

We have presented a novel approach that uses visual analysis to help identifying and prioritizing code-quality issues based on enhanced circular bundle views. The technique combines several important indicators for causes of increased maintenance costs, namely code complexity, frequent and recent changes to the respective code, and entanglement among modules. Indicators are encoded in a way that is easy to assess and understand. Even for non-experts in software engineering, the visual representation facilitates a general understanding of the underlying processes and available decisions. We have evaluated our approach by means of a case study conducted on an industrial-size software system. The case study showed that our method is scalable and, in conjunction with the used indicators, yields added value. Other indicators can as well be used in a canonical way.

As future work, we plan to add evolution analysis of metric values. By that, unusual change rates can be identified. Furthermore, we want to introduce indicators other than file-related software metrics, such as attributing edges as well: The more (upward) layers an edge crosses to reach its destination, the more suspicious it is. Hence, by measuring an edge's suspiciousness, we could emphasize suspicious edges visually. As another example, depicting test coverage for each module might point users towards untested code that is potentially more error-prone. Moreover, we plan to improve occlusion handling, e.g., by implementing a transparency lens that only affects hovered parts of a module instead of the entire module. By conducting controlled experiments, we plan to further investigate the benefits of our approach.

Acknowledgements

We would like to thank *Francotyp Postalia GmbH* for providing their code base and performing the case study with us. This work was supported by the ZIM program of the Federal Goverment of Germany (BMWi).

References

[1] *IEEE Standard Glossary of Software Engineering Terminology 610.12-1990*, 1990.

[2] Sazzadul Alam and Philippe Dugerdil. Evospaces: 3d visualization of software architecture. In *Proc. SEKE*, pages 500–505, 2007.

[3] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. *J VLC*, 6:119–133, 1995.

[4] Michael Balzer, Oliver Deussen, and Claus Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proc. SOFTVIS*, pages 165–172, 2005.

[5] Johannes Bohnet and Jürgen Döllner. Monitoring code quality and development activity by software maps. In *Proc. MTD*, pages 9–16, 2011.

[6] Fabrice Bourquin and Rudolf K. Keller. High-impact refactoring based on architecture violations. In *Proc. CSMR*, pages 149–158, 2007.

[7] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[8] Andrea Capiluppi, Alvaro E. Faria, and Juan F. Ramil. Exploring the relationship between cumulative change and complexity in an open source system. In *Proc. CSMR*, pages 21–29, 2005.

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE TSE*, 20:476–493, June 1994.

[10] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. ICPC*, pages 49–58, 2007.

[11] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE TSE*, 28:396–412, April 2002.

[12] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE TSE*, 27(1):1–12, January 2001.

[13] N. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 1997.

[14] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: The use of color and

third dimension. In *Proc. ICSM*, pages 99–108. IEEE Computer Society, 1999.

[15] Jane Huffman Hayes, Sandip C. Patel, and Liming Zhao. A metrics-based software maintenance effort model. In *Proc. CSMR*, pages 254–258, 2004.

[16] Christopher G. Healey, Kellogg S. Booth, and James T. Enns. High-speed visual estimation using preattentive processing. *ACM Transactions on Human-Computer Interaction*, 3(2):107–135, 1996.

[17] Sallie Henry and Steve Wake. Predicting maintainability with software quality metrics. *J SOFTW MAINT RE-PR*, 3(3):129–143, 1991.

[18] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG*, 12:741–748, 2006.

[19] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proc. ASE*, pages 214–223, 2005.

[20] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE TSE*, 29(9):782–795, 2003.

[21] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[22] Claus Lewerentz, Frank Simon, and Frank Steinbrückner. Crococosmos. In *Graph Drawing*, volume 2265 of *LNCS*, pages 72–76. Springer, 2002.

[23] S. Muthanna, K. Ponnambalam, K. Kontogiannis, and B. Stacey. A maintainability model for industrial software systems using design level metrics. In *Proc. WCRE*, pages 248–256, 2000.

[24] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3d visualizations of hierarchical information. In *Proc. CHI*, pages 189–194. ACM, 1991.

[25] Samuel Silva, Joaquim Madeira, and Beatriz Sousa Santos. There is more to color scales than meets the eye: A review on the use of color in visualization. In *Proc. IV*, pages 943–950. IEEE Computer Society, 2007.

[26] Yogesh Singh, Pradeep Kumar Bhatia, and Omprakash Sangwan. Predicting software maintenance using fuzzy model. *ACM SIGSOFT*, 34:1–6, July 2009.

[27] Alexandru Telea and Lucian Voinea. Case study: Visual analytics in software product assessments. In *Proc. VISSOFT*, pages 65–72, 2009.

[28] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, May 2001.

[29] Bogdan Vasilescu, Alexander Serebrenik, and Mark van den Brand. You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In *Proc. ICSM*, pages 313–322, 2011.

[30] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, 2nd edition, 2004.

[31] Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In *Proc. SOFTVIS*, pages 155–164, 2008.

[32] M. Wilhelm and S. Diehl. Dependency viewer - a tool for visualizing package design quality metrics. In *Proc. VISSOFT*, pages 34–35, 2005.

This is a preprint version of the paper. The official print version can be obtained from the IEEE Computer Society. When citing the paper, please use the following BibTeX entry:

```
@inproceedings{TBD2012,
author = { Jonas Trümper and Martin Beck and Jürgen Döllner },
title = { A Visual Analysis Approach to Support Perfective Software
Maintenance },
booktitle = { Proceedings of the 16th International Conference on
Information Visualisation },
year = { 2012 },
publisher = { IEEE Computer Society }
}
```