Know Thy Source Code

Is it mostly dead or alive?

Vladimir Markovets
Institute of Informatics
University of Warsaw
v.markovets@uw.edu.pl

Robert Dąbrowski
Institute of Informatics
University of Warsaw
r.dabrowski@mimuw.edu.pl

Grzegorz Timoszuk
Institute of Informatics
University of Warsaw
g.timoszuk@mimuw.edu.pl

Krzysztof Stencel
Institute of Informatics
University of Warsaw
stencel@mimuw.edu.pl

ABSTRACT

Nowadays, even small systems contain numerous components with complex dependencies. These components differ in importance and quality. Some of them get deprecated over time and can be removed from the project. This leads to the question relevant for all architects: which parts of my source code are still alive? To answer this question we harness the graph-based model of software. We perform a dynamic analysis of internal behaviour of components by eardropping the control flow. We then augment the graph model of the system with the results of this analysis and visualize it. We render execution paths over the nodes and their connections. The vividness of colours reflects the frequency of calls. This visualization helps judging at a glance which parts of software remained unvisited during software execution. Probably those parts are actually dead, thus should absorb little further maintenance, if any. In this paper we describe proof-of-concept tools to support this approach and report results of analysis of selected open-source Java projects of various sizes.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.7 [**Software Engineering**]: Maintainability—*maintenance management,measurement*

Keywords

architecture, run-time analysis, execution

1. INTRODUCTION

The complexity of software systems and their development processes has rapidly grown in recent years. For many systems their architectures have actually become hardly manageable. On the other hand development is performed by teams that change over time, work under pressure, with incomplete documentation and continually changing requirements. Multiple development technologies, programming languages, coding standards, partial releases make this situation even more severe. To ensure that this environment remains predictable, the role of software architects has been introduced. They are obliged to control the whole source code. One of the missions of software engineering is to help them in their efforts.

In our research we focus on assessment, comprehension and analysis of software architectures. As one of the research tasks we address the question frequently asked by architects: which parts of my source code are still alive? In this paper we demonstrate our vision of how to address this question. It can be summarized as follows. All software system artefacts and all software engineering process artefacts created during a software project are explicitly organized in an architecture warehouse [3] as vertices of a graph connected by multiple edges that represent numerous kinds of dependencies among those artefacts.

Graphs can be visualised using well-known drawing techniques and algorithms. We employ those techniques and extend them to visualise relative importance of components by the size of nodes, quality of components by colours and the coupling of components by the density of the connections [4, 5, 6]. In particular, a more important node is rendered as a bigger dot. Currently, we measure importance of components using PageRank. However, it can be substituted by any other metric. We use green to mark components of good quality, while red indicates poor quality. We measure quality using CodePro Analytix toolset [10]. As with importance, also the quality measure can be arbitrarily chosen. We depict complexity of the source code by explicitly showing static dependencies among the components. We also eardrop software execution and collect information on dynamic dependencies among the components. We portray runtime calls using yellow edges. Their thickness represents the frequency of calls. We capture logs of software execution using Kieker [20]. Nevertheless, we can use any other runtime monitoring framework that is aware of software internal representation.

The paper is organised as follows. In Section 2 we address the related work. In Section 3 we recall the graph-based model for representing architectural knowledge. In Section 4 we present results and possible extensions. Section 5 concludes.

2. MOTIVATION AND RELATED WORK

The idea described in this paper has been contributed to by several existing approaches and practices.

Multiple graph-based models have been proposed to reflect architectural facets, e.g. to represent architectural decisions and changes [21], to discover implicit knowledge from architecture change logs [19] or support architecture analysis and tracing [2]. Graph-based models have also become helpful in UML model transformations, especially in model driven development (MDD) [7]. Automated transition (e.g. from use cases to activity diagrams) have been considered [13], along with additional traceabilities that could be established through automated transformation, e.g. relate requirements to design decisions and test cases. An approach to automatically generate activity diagrams from use cases while establishing traceability links has already been implemented (RAVEN).

Architectural knowledge can also be extended by data gathered during software executions [20]. Aspects of tracing architectural decision to requirements has been thoroughly investigated in [1, 18, 8]. Analysis on gathering, managing and verifying architectural knowledge has been conducted and presented in [12]. Changes made in architecture management during last twenty years has been summarised in the survey [9].

Visualization of software architecture has been a research goal for years. Tools like Bauhaus [11], Source Viewer 3D [15], JIVE [17], code_smarm [16] and StarGate [14] are interesting attempts in visualization. However none of them simultaneously supports aggregation (e.g. package views), drill-down, picturing the code quality and dependencies. Moreover, all of them are significantly more complex when compared to our proposal.

3. GRAPH MODEL

We recall the theoretical model [4] for unified representation of architectural knowledge. Definition of the model is based on directed labelled multigraph. According to the model, the *software architecture graph* is an ordered triple $(\mathcal{V}, \mathcal{L}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices that reflect all artifacts created during a software project, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{L} \times \mathcal{V}$ is the set of directed edges that represent dependencies (relations) among those artifacts, and $\mathcal{L}$ is the set of labels which qualify the artifacts and their dependencies. There can be more than one edge from one vertex to another vertex (multigraph), as artifacts can be in more than one relation. The relations are typically not reflexive (the graph is directed). Vertices of the project graph are created when artifacts are produced during software development process. By default vertices represent parts of the source code, like: modules, classes, methods, tests (e.g. unit tests, integration tests, stress tests). Other examples of vertices are: documents (e.g. requirements, use cases, change requests), coding guidelines, source codes in higher level languages (e.g. yacc grammars), configuration files, make files or build recipes, tickets in issue tracking systems etc.

Our graph model is general and scalable, fits both small and huge projects, and has been tested in practice [5]. The tests proved that in case of a large project it becomes too complex to be human-tractable as a whole. This has confirmed that some rules for graph transformation are a must, since software architects are interested both in an overall (top-level) picture and in particular (low-level) details, e.g. in details that satisfy some additional restrictions.

Intuitions for such transformations are e.g.: return the subgraph including only methods that call the given method; return the subgraph of all public methods for the given class (either including or excluding inherited ones). Obviously, in the graph model the queries that compute such transformations are computationally inexpensive, as usually they only need to traverse the graph (or even its small fraction).

4. MAIN RESULT

In this Section we describe two possible scenarios of dynamic analysis to identify dead code. We discuss experimental results gathered for two projects: JUnit and JLoxim. JUnit is a well known Java library. JLoxim is an experimental semi-structured database system.

We focus on dynamic analysis of software conducted in two types of scenarios: (1) running software tests (test suites for unit tests, integration tests, etc.); (2) observing production execution of a software system. Data gathered during different scenarios vary and have different purpose.

4.1 Software testing

Dynamically gathered information during tests helps understanding how the control is passed among software artefacts. We can assess method coverage, not only metrics of coverage in terms of lines of code. The method coverage is important especially in case of integration testing. Our approach works well in case of properly defined testing scenarios. The method coverage empowered by statically gathered information (including artefact's importance and quality) helps architects judging if the most important (big dots) or vulnerable (red dots) artefacts are properly tested. Dynamically gathered data also reveals connections of artefacts that are impossible or extremely hard to discover during static analysis. In particular, for software systems relying on reflection, inversion of control or aspect oriented programming dynamic analysis is crucial for proper assessment of software architecture. The dynamic analysis of control flow may also reveal prohibited actions e.g. references to forbidden modules introduced by reflection mechanism.

4.2 Software execution

Data gathered during dynamic analysis of program runtime helps identifying artefacts and flows heavily used in real-life program execution. Although certain scenarios are theoretically possible and worth considering, they may never occur in practice. Dynamic data combined with information about importance and quality precisely points the parts of the software that are most crucial. For example, frequently used important modules of poor quality require immediate attention

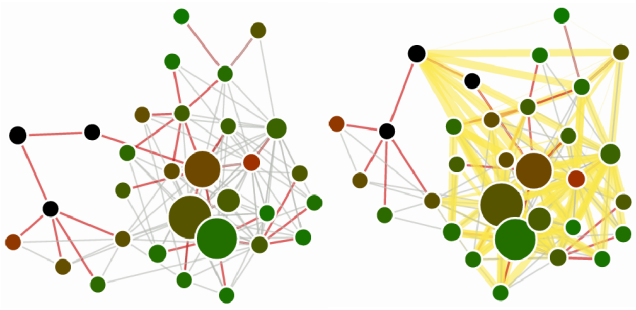

Figure 1: The visualisations of data on JUnit collected by static analysis only and by both static and dynamic analyses

of the software architect. It may also help new developers understanding the system they begin to work with. Moreover, regular snapshots of dynamic analyses may be helpful during identification of software usage schemata that change over time. For example, they can help understanding system performance degradation or identifying bottlenecks.

4.3 Results of experiments

In this Section we describe the results of experiments. We analysed two systems statically and dynamically. We generated software visualisations that helped quickly finding the dead code. We present two cases: (1) JUnit with dynamic analysis gathered during *software testing*, and (2) JLoxim with dynamic analysis gathered during *software execution*. We visualise packages and classes with relations between them. Red edges present containment of packages. Grey edges are import relations. Yellow edges correspond to calls between artefacts. Thick lines mark places where heavy communication takes place.

4.3.1 JUnit

Figure 1 presents two states of the architecture warehouse on JUnit. The picture on the left contains data discovered during static analysis only. The picture on the right is the result of adding data from dynamic analysis of a run of the internal Junit test suite. The data from dynamic analysis reveals new dependencies. This is in line with the conclusion that can be drawn after careful analysis of the source code—JUnit relies strongly on reflection. Moreover, the project as a whole has good quality (no big red dots). There is a medium size red dot in the centre-right part of the right picture in Figure 1. This artefact should be analysed in detail by the software architect as it is frequently used. On the other hand, there are red dots in left part of the picture that are practically never used during systematic tests. They are suspects of being dead. Finally we see that most of packages use each other. In case of a small project like JUnit this is not a severe problem. However, in case of a big project this property should draw attention of the software architect.

4.3.2 JLoxim

Figure 2 presents the data gathered using static analysis of JLoxim. Figure 3 shows the same data augmented by data collected during a sample production execution of JLoxim. There is significantly less connections between modules in

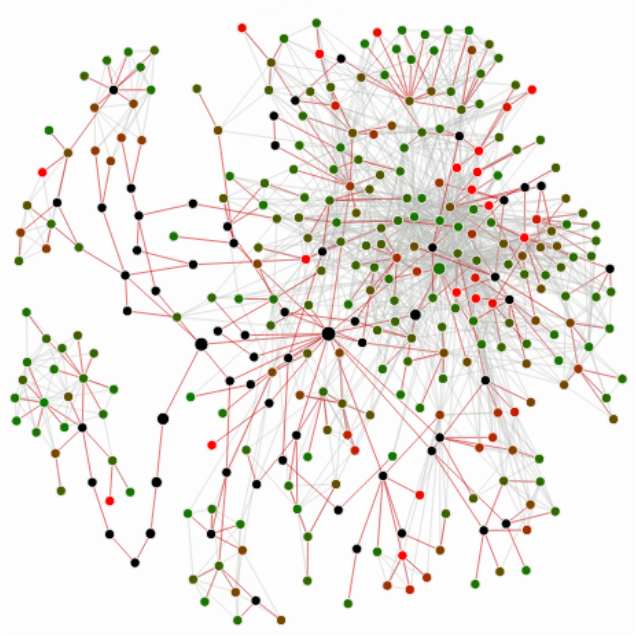

Figure 2: The visualisation of data on JLoxim collected by static analysis only

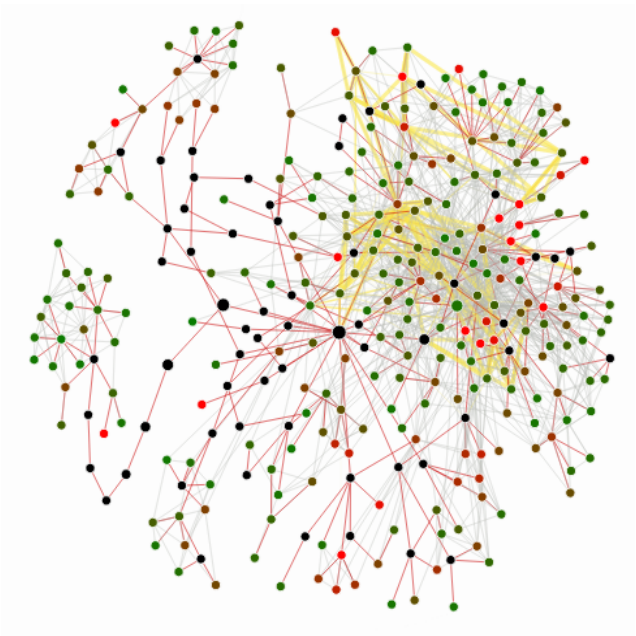

Figure 3: The visualisation of data on JLoxim collected by both static and dynamic analyses

comparison with JUnit. This leads to a conclusion that JLoxim is properly modularised. On the other hand, Figure 3 shows that the query execution relies on packages having poor quality. Additionally, there are many connections in a small part of the system (observe the central part of Figure 3). There are multiple yellow lines there that indicate many calls. This should be a clear hint for the software architect that the query processing module needs more attention and probably requires refactorisation. Moreover, we notice that poor quality packages frequently use one another.

5. CONCLUSION

In this paper we focused on evaluating how data gathered during dynamic analysis of a software system influences the assessment of its architecture. In our methodology we employ the architecture warehouse that contains knowledge on artefacts and their relationships gathered during static analysis of the source code. We augment that data by including information about calls that actually took place during runtime. More precisely, we record the number of observed calls between artefacts. This has proven helpful in further analysis. Eventually, we visualize the data collected in the software architecture warehouse. This allows architects and developers quickly assessing and comprehending the software architecture.

In particular, such information and visualisations facilitate quick assessment of unused parts of the source code. It may concern both the test execution and the production execution. In general, our method of visualisation becomes especially handy in case of a software that performs multiple actions as side effects, and in case of a large software system. For such architectures, clear presentation of information about dynamics of calls becomes a must. We use sizes of nodes to depict importance, the colour of nodes to show quality, and the thickness of connections to render the frequency of calls observed at the runtime.

Experiments conducted using this approach are promising. We plan to perform additional evaluation using different projects, both with respect to test coverage and productive execution. As the result of such experiments new features useful for architects will certainly arise.

6. REFERENCES

[1] P. Avgeriou, J. Grundy, J. G. Hall, P. Lago, and I. Mistrík, editors. *Relating Software Requirements and Architectures*. Springer, 2011.

[2] H. P. Breivold, I. Crnkovic, and M. Larsson. Software architecture evolution through evolvability analysis. *Journal of Systems and Software*, 85(11):2574–2592, 2012.

[3] R. Dabrowski. On architecture warehouses and software intelligence. In T.-H. Kim, Y.-H. Lee, and W.-C. Fang, editors, *FGIT*, volume 7709 of *Lecture Notes in Computer Science*, pages 251–262. Springer, 2012.

[4] R. Dabrowski, K. Stencel, and G. Timoszuk. Software is a directed multigraph. In I. Crnkovic, V. Gruhn, and M. Book, editors, *ECSA*, volume 6903 of *Lecture Notes in Computer Science*, pages 360–369. Springer, 2011.

[5] R. Dabrowski, K. Stencel, and G. Timoszuk. Improving software quality by improving architecture management. In B. Rachev and A. Smrikarov, editors, *CompSysTech*, pages 208–215. ACM, 2012.

[6] R. Dabrowski, K. Stencel, and G. Timoszuk. One graph to rule them all - software measurment and management. In L. Popova-Zeugmann, editor, *CS&P*, volume 928 of *CEUR Workshop Proceedings*, pages 79–90. CEUR-WS.org, 2012.

[7] J. Derrick and H. Wehrheim. Model transformations across views. *Sci. Comput. Program.*, 75(3):192–210, 2010.

[8] A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *ASE*, pages 163–171. IEEE Computer Society, 2002.

[9] D. Garlan and M. Shaw. Software architecture: reflections on an evolving discipline. In T. Gyimóthy and A. Zeller, editors, *SIGSOFT FSE*, page 2. ACM, 2011.

[10] Google. CodePro Analytix. `https://developers.google.com/java-dev-tools/codepro/doc/`.

[11] R. Koschke. Software visualization for reverse engineering. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 138–150. Springer, 2001.

[12] P. Kruchten. Where did all this good architectural knowledge go? In M. A. Babar and I. Gorton, editors, *ECSA*, volume 6285 of *Lecture Notes in Computer Science*, pages 5–6. Springer, 2010.

[13] T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, editors. *Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings*, volume 6138 of *Lecture Notes in Computer Science*. Springer, 2010.

[14] K.-L. Ma. Stargate: A unified, interactive visualization of software projects. In *PacificVis*, pages 191–198. IEEE, 2008.

[15] J. I. Maletic, A. Marcus, and L. Feng. Source viewer 3d (sv3d) - a framework for software visualization. In L. A. Clarke, L. Dillon, and W. F. Tichy, editors, *ICSE*, pages 812–813. IEEE Computer Society, 2003.

[16] M. Ogawa and K.-L. Ma. code_swarm: A design study in organic software visualization. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1097–1104, 2009.

[17] S. P. Reiss. Dynamic detection and visualization of software phases. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–6, 2005.

[18] G. Spanoudakis and A. Zisman. Software traceability: a roadmap. *Handbook of Software Engineering and Knowledge Engineering*, 3:395–428, 2005.

[19] A. Tang, P. Liang, and H. van Vliet. Software architecture documentation: The road ahead. In *WICSA*, pages 252–255, 2011.

[20] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In D. R. Kaeli, J. Rolia, L. K. John, and D. Krishnamurthy, editors, *ICPE*, pages 247–248. ACM, 2012.

[21] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A graph based architectural (re)configuration language. In *ESEC / SIGSOFT FSE*, pages 21–32, 2001.