Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1473–1476

Magnify – a new tool for software visualization

Cezary Bartoszuk, Grzegorz Timoszuk, Robert Dąbrowski, Krzysztof Stencel
Institute of Informatics
University of Warsaw
Banacha 2, 02-097 Warsaw, Poland

***Abstract*—Modern software systems are inherently complex. Their maintenance is hardly possible without precise up-to-date documentation. It is often tricky to document dependencies among software components by only looking at the raw source code. We address these issues by researching new software analysis and visualization tools.**

In this paper we focus on software visualisation. *Magnify* is our new tool that performs static analysis and visualization of software. It parses the source code, identifies dependencies between code units and records all the collected information in a repository based on a language-independent graph-based data model. Nodes of the graph correspond to program entities of disparate granularity: methods, classes, packages etc. Edges represent dependencies and hierarchical structure. We use colours to reflect the quality, sizes to display the importance of artefacts, density of connections to portray the coupling. This kind of visualization gives bird's-eye view of the source code. It is always up to date, since the tool generates it automatically from the current revision of software. In this paper we discuss the design of the tool and present visualizations of sample open-source Java projects of various sizes.

I. Introduction

THE complexity of software systems and their development processes have rapidly grown in recent years. In numerous companies the high level structure of dependencies between software components is kept only in the heads of developers. This makes the development process fragile, since teams composed of humans are inherently volatile. Therefore, development teams need methods and tools to inspect *current* states of complex systems and their fragments.

In this paper we describe a software visualization tool *Magnify* that caters for these needs. It shows top level views of software entities of disparate granularities: systems, components, modules, classes etc. These views contain graphic presentation of the importance, the quality and the coupling of subcomponents.

Given a source code bundle *Magnify* parses and analyzes it in order to create a graph-based model [1]. This model is then persisted in the architecture warehouse that provides data for software intelligence [2]. One of its methods is visualization. *Magnify* reads the data from the warehouse and produces a graph laid out on a plain. Nodes of the graph are entities (classes, packages etc.) of the provided source code.

The size of a node reflects the importance of the corresponding artefact. In the current version of *Magnify* we use PageRank to assess importance. The colour of a node represents the quality of the corresponding piece of code. In the examples presented in this paper, we use the numbers of lines per class. Any software metrics accompanied with threshold values for green and red can be employed.

Edges represent relationships of two kinds: structural inclusion and dependency.

The paper is organised as follows. In Section II we address the related work. In Section III we recall the theoretical foundations for our research. In Section IV we describe the design of *Magnify* and possible extension points in its architecture. In Section V we present visualizations of sample open-source Java projects of various sizes. Section VI concludes.

II. Motivation

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

A unified approach to software systems and software processes has already been presented in [3]. Software systems were perceived as large, complex and intangible objects developed without a suitably visible, detailed and formal descriptions of how to proceed. It was suggested that software process should be included in software project as parts of programs with explicitly stated descriptions; software architect should communicate with developers, customers and other managers through software process programs indicating steps that are to be taken in order to achieve product development or evolution goals.

Multiple graph-based models have been proposed to reflect architectural facets, e.g. to represent architectural decisions and changes [4], to discover implicit knowledge from architecture change logs [5] or support architecture analysis and tracing [6]. Graph-based models have also become helpful in UML model transformations, especially in model driven development (MDD) [7].

Visualization of software architecture has been a research goal for years. The tools like Bauhaus [8], Source Viewer 3D [9], Gevol [10], JIVE [11], evolution radar [12], code_smarm [13] 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. In our opinion noteworthy better effects can be achieved with simpler facilities. Movies and the third dimension are not necessary to quickly assess the quality, robustness and resilience of an architecture.

978-1-4673-4471-5/$25.00 © 2013, IEEE

III. Theoretical foundations

A. Model

We recall the theoretical model [1] 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 artefacts 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 artefacts, and $\mathcal{L}$ is the set of labels which qualify the artefacts and their dependencies.

Example 1. Each artefact can be described by a set of labels. A method can be described by labels showing that it is a part of project source code (*code*); written in Java (*java*); its revision is 456 (*r:456*); it is *abstract* and *public*. Edges are directed and may have multiple labels as well, e.g.: a package *contains* a class; a method *calls* another method.

The transformations and metrics recalled below give the foundation for the layer of *software intelligence* tools [2].

B. Transformations

Our graph model is general and scalable, fits both small and huge projects [15], and has been tested in practice [16].

The tests proved that in case of a large project its graph model is too complex to be human-tractable as a whole. This has confirmed that transformations and views of the graph model are a must.

Example 2. For a given software graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ and a subset of its labels $\mathcal{L}' \subseteq \mathcal{L}$, its *filter* is a transformation $\mathcal{G}|_{\mathcal{L}'} = (\mathcal{V}', \mathcal{E}', \mathcal{L}')$ where $\mathcal{V}'$ and $\mathcal{E}'$ have a label in $\mathcal{L}'$.

C. Metrics

For complex projects their quantitative evaluation is a must. The graph-based approach is in line with best practices for metrics [17], [18], allows for easy translation of existing metrics into graph terms [19], ensures they can be efficiently calculated using graph algorithms. It also allows designing new metrics that combine both software system and software process artefacts [20].

Example 3. For a given software graph $\mathcal{G} = (\mathcal{V}, \mathcal{L}, \mathcal{E})$, its *metric* is a transformation $m : \mathcal{G} \mapsto \mathcal{R}$ where $\mathcal{R}$ denotes real numbers and m can be effectively calculated by a graph algorithm on G.

IV. Magnify

We implemented *Magnify* as a server system, based on a graph database, with web front-end written in Scala. *Magnify* functionality allows loading source code bundles, analyzing them and displaying the resulting graphs of software components. Figure 1 shows a sample of *Magnify*'s GUI.

The analytical backend is composed of three main modules: the parser, the graph storage and the analysis engine.

The parser is the only part of *Magnify* that must be programming language specific. Its responsibility is to transform

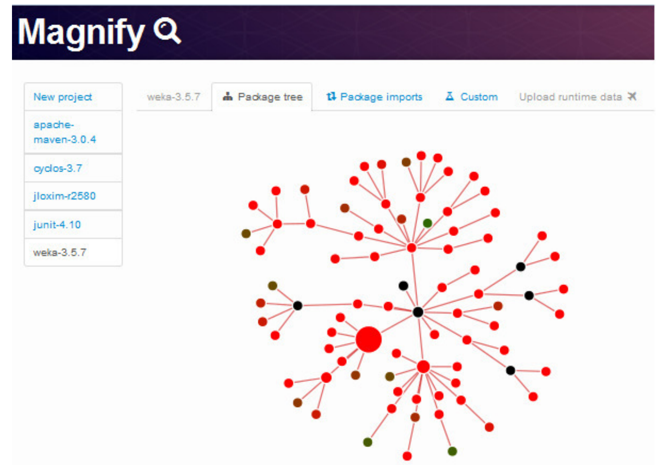

Fig. 1. *Magnify* functionality - main view

contents of a source code bundle into an abstract software graph. This graph is then persisted. Currently the implementation contains a Java 5 parser based on the `javaparser` library. The graph storage is based on Tinkerpop Blueprints specification. When the source code is converted and stored in the abstract language-agnostic form, the analysis engine will run. The implementation computes PageRank of every node in the graph and code quality metrics for classes. These metrics are then escalated to the package level. Figure 2 shows the referential deployment diagram of *Magnify*.

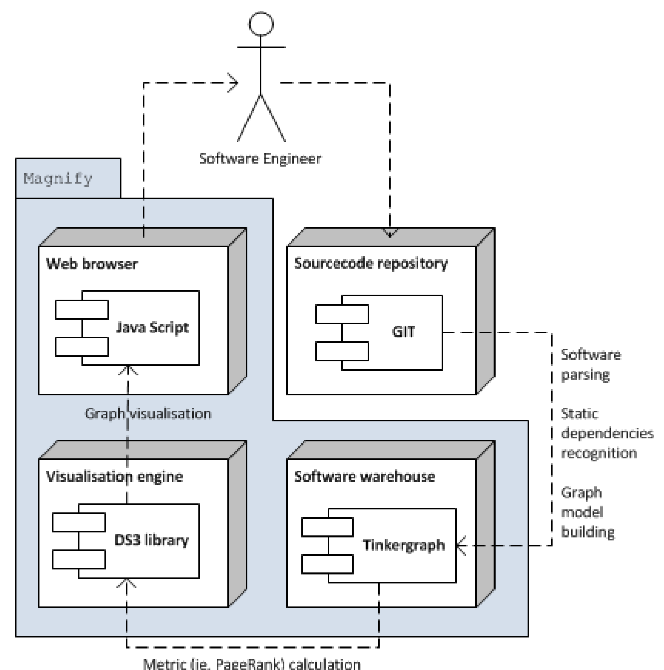

Fig. 2. General architecture of *Magnify*

The architecture of *Magnify* is flexible enough to replace any of its components and add new constituents. As noted

above, we can replace the repository with any graph database that conforms to Blueprints API. New data providers can be added, like parsers for more programming languages, runtime profilers, analyzers of version control systems, and analyzers of web data (e.g. forum discussions).

V. Evaluation

In this Section we show sample visualizations produced by *Magnify*. We have chosen five open-source projects that significantly vary in size and quality. All of them have a noteworthy number of users. They are well adopted by the software development community.

These are Cyclos, Play, Spring and Karaf. For each system we present its top-level visualization created by *Magnify*. Then, we analyze the resulting images and enumerate conclusions that can be drawn from them.

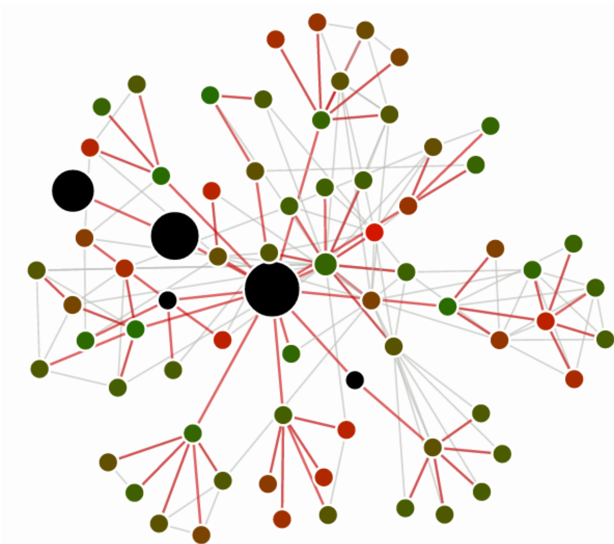

Fig. 3. The visualization of Spring context 3.2.2 produced by *Magnify*

A. Spring context 3.2.2

Spring is one of the most popular enterprise application frameworks in the Java community. It provides an infrastructure for dependency injection, cache, transactions, data base access and many more. Figure 3 shows the visualization of Spring produced by *Magnify*.

The structure of dependencies implies that Spring is well designed. The graph is notably sparse. The only packages detected as important are empty vendor packages. All the packages that do contain classes are of the same importance. This indicates a well balanced software. Figure 3 contains no brightly red packages. This means that on average the classes are small in most of packages. Thus, the overall quality is satisfactory.

B. Cyclos 3.7

Cyclos is a complete on-line payment system. Figure 4 presents visualization of this system produced by the *Magnify* tool.

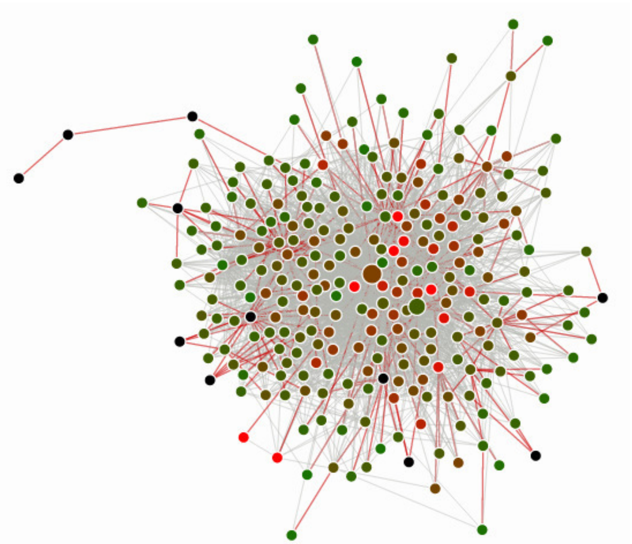

Fig. 4. The visualization of Cyclos 3.7 produced by *Magnify*

Unfortunately, this time the dependency graph is exceptionally dense. The software engineering experience indicates that the development and maintenance of software systems with so tight coupling is difficult, costly and error-prone. On the other hand, Figure 4 shows few packages in which classes are big on average. That means that overall complexity of the classes themselves is acceptable.

Cyclos is a profound example of a system that should be split into orchestrated group of communicating systems. This kind of refactoring will significantly improve the quality of this software. It will also reduce the cost of further development and maintenance.

C. Play 1.2.5

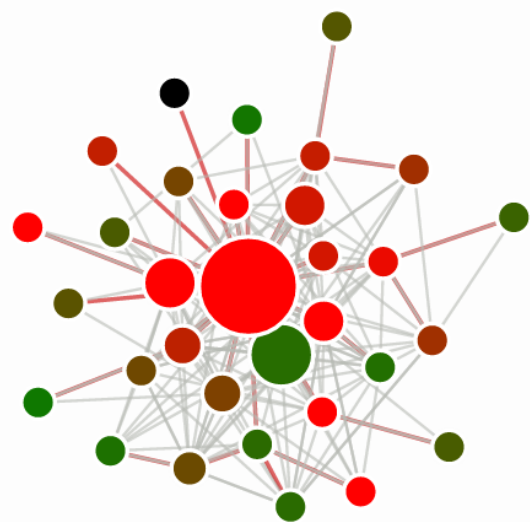

Fig. 5. The visualization of Play 1.2.5 produced by *Magnify*

Play is a popular Scala and Java web framework. Figure 5 shows the visualization of Play using *Magnify*.

It presents a small project with decent amount of dependencies. The flat package structure is typical for dynamic languages. The biggest node corresponds to the project root package `play`. Brightly red packages reveal potentially high complexity of their classes.

D. Apache Karaf 3.0.0 RC1

Apache Karaf is a small OSGi container to deploy various components and applications. Even though it is split into many packages, the number of dependencies is small. Many subtrees of the package hierarchy have only a single dependency on the rest of the system. Thus, Karaf is well packaged.

Figure 6 shows that overall code quality in Karaf is good. There are only a few packages where the average class size is alarming.

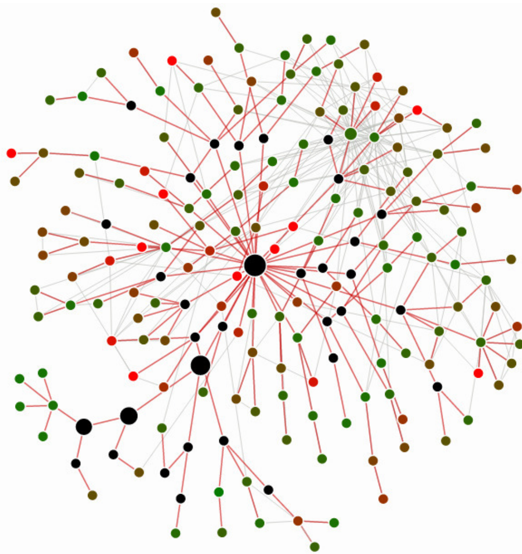

Fig. 6. The visualization of Karaf 3.0.0-RC1 produced by *Magnify*

VI. Conclusion

We follow the research on anayslis and visualisation of software and software process, and promote an approach that avoids separation between software and software process artefacts. We demonstrate that the implementation of such approach is feasible. We implement software intelligence on top of a software warehouse based on our theoretical graph-based model. We execute experiments on open-source Java programs using those tools.

In this paper we presented *Magnify* - a tool that performs static analysis and visualization of software systems. It focuses on relationships between components rather than on their internal structure.

Magnify is a general tool that can adapt other quality metrics and importance estimates. Flexibility of its design allows replacing any of its components and adding new parts. In order to support the analyses for another programming language, we have to add only an appropriate parser. All other facilities (the repository and analytic algorithms) need not be changed.

Promising ideas worth implementing in the near future include: (1) improving the vertex clustering algorithm for module repackaging, (2) gathering the information across revisions and (3) adding metadata stating how packages are split into modules and how these modules depend on each other. This kind of metadata would constitute a specification that can be matched against the source code.

References

[1] R. Dąbrowski, K. Stencel, and G. Timoszuk, "Software is a directed multigraph," in *ECSA*, ser. Lecture Notes in Computer Science, I. Crnkovic, V. Gruhn, and M. Book, Eds., vol. 6903. Springer, 2011, pp. 360–369.

[2] R. Dąbrowski, "On architecture warehouses and software intelligence," in *FGIT*, ser. Lecture Notes in Computer Science, T.-H. Kim, Y.-H. Lee, and W.-C. Fang, Eds., vol. 7709. Springer, 2012, pp. 251–262.

[3] L. J. Osterweil, "Software processes are software too," in *ICSE*, W. E. Riddle, R. M. Balzer, and K. Kishida, Eds. ACM Press, 1987, pp. 2–13.

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

[5] A. Tang, P. Liang, and H. van Vliet, "Software architecture documentation: The road ahead," in *WICSA*, 2011, pp. 252–255.

[6] H. P. Breivold, I. Crnkovic, and M. Larsson, "Software architecture evolution through evolvability analysis," *Journal of Systems and Software*, vol. 85, no. 11, pp. 2574–2592, 2012.

[7] J. Derrick and H. Wehrheim, "Model transformations across views," *Sci. Comput. Program.*, vol. 75, no. 3, pp. 192–210, 2010.

[8] R. Koschke, "Software visualization for reverse engineering," in *Software Visualization*, ser. Lecture Notes in Computer Science, S. Diehl, Ed., vol. 2269. Springer, 2001, pp. 138–150.

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

[10] C. S. Collberg, S. G. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *SOFTVIS*, S. Diehl, J. T. Stasko, and S. N. Spencer, Eds. ACM, 2003, pp. 77–86, 212–213.

[11] S. P. Reiss, "Dynamic detection and visualization of software phases," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–6, 2005.

[12] M. D'Ambros, M. Lanza, and M. Lungu, "The evolution radar: visualizing integrated logical coupling information," in *MSR*, S. Diehl, H. Gall, and A. E. Hassan, Eds. ACM, 2006, pp. 26–32.

[13] M. Ogawa and K.-L. Ma, "code_swarm: A design study in organic software visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 6, pp. 1097–1104, 2009.

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

[15] P. Tabor and K. Stencel, "Stream execution of object queries," in *FGIT-GDC/CA*, 2010, pp. 167–176.

[16] R. Dąbrowski, K. Stencel, and G. Timoszuk, "Improving software quality by improving architecture management," in *CompSysTech*, 2012, pp. 208–215.

[17] F. Abreu and R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *Proceedings of the 4th International Conference on Software Quality*, 1994.

[18] J. M. Roche, "Software metrics and measurement principles," *SIGSOFT Softw. Eng. Notes*, vol. 19, pp. 77–85, January 1994. [Online]. Available: http://doi.acm.org/10.1145/181610.181625

[19] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, June 1994. [Online]. Available: http://portal.acm.org/citation.cfm?id=630808.631131

[20] R. Dąbrowski, G. Timoszuk, and K. Stencel, "One graph to rule them all - software measurment and management," *Fundamenta Informaticae*, vol. to appear, 2013.