Visual Exploration of Large-Scale Evolving Software

Richard Wettel
REVEAL@Faculty of Informatics
University of Lugano, Switzerland

Abstract

The comprehensive understanding of today's software systems is a daunting activity, because of the sheer size and complexity that such systems exhibit. Moreover, software systems evolve, which dramatically increases the amount of data one needs to analyze in order to gain insights into such systems. Indeed, software complexity is recognized as one of the major challenges to the development and maintenance of industrial-size software projects.

Our vision is a 3D visualization approach which helps software engineers build knowledge about their systems. We settled on an intuitive metaphor, which depicts software systems as cities. To validate the ideas emerging from our research, we implemented a tool called CodeCity. We devised a set of visualization techniques to support tasks related to program comprehension, design quality assessment, and evolution analysis, and applied them on large open-source systems written in Java, C++, or Smalltalk. Our next research goals are enriching our metaphor with meaningful representations for relations and encoding higher-level information.

1. Introduction

Understanding today's software systems is a daunting and costly activity. While software maintenance claims up to 90% of the cost of a software system [7], more than half of the time dedicated to it is spent in program comprehension tasks [21]. This is due to the complexity of the systems, *i.e.,* one of the major challenges of industrial-size software, and to their sheer size. Moreover, because of the evolutionary nature of software [12], the amount of data one needs to analyze when it comes to several versions of such a system is overwhelming. In this context, tool support is essential. An efficient means for synthesizing large amounts of data and for building a mental model of a software system is visualization. We propose an integrated software analysis environment based on a city metaphor, described next.

2. The City Metaphor

In the context of the EvoSpaces[1] project, which aims at exploiting multi-dimensional navigation spaces to visualize evolving software, we have experimented with several metaphors [3] to provide some tangibility to the abstract nature of software. We settled on a 3D *city* metaphor [17], for it confers a complex exploratory environment with a clear notion of locality, which counteracts disorientation (an open challenge in 3D visualization). This led to the adoption of the metaphor in the project's supporting tool [1].

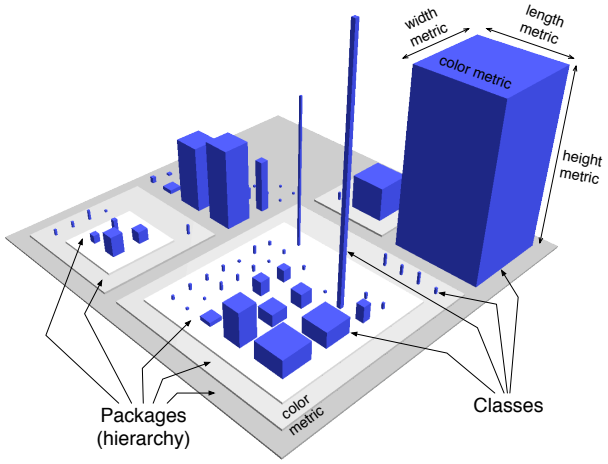

Figure 1. Principles of our city metaphor

We represent classes as buildings and packages as the districts in which the buildings reside (See Figure 1). The visual properties of the city artifacts depict metric values. Our typical configuration is: for classes, the number of methods (NOM) metric mapped on the buildings' height and the number of attributes (NOA) on their base size, and for packages the nesting level mapped on the districts' color saturation (*i.e.,* root packages are dark, while deeply nested packages are light-colored districts). The package hierarchy is thus reflected by the city's district structure.

[1] http://www.inf.unisi.ch/projects/evospaces

3. Approach and Validation

We applied our approach in 3 contexts: program comprehension, evolution analysis, and design quality assessment.

3.1. Program Comprehension

We conducted a program comprehension experiment, described in detail in [16], on ArgoUML, a 140 kLOC Java system. The overview of the city of ArgoUML shows the system's structure and points out the outliers in terms of the mapped metrics (Figure 2). We identified three archetypes of prominent buildings: skyscrapers (*i.e.,* classes with many methods and few attributes), parking lots (*i.e.,* classes with few methods and many attributes), and massive buildings (*i.e.,* classes with both many methods and attributes).

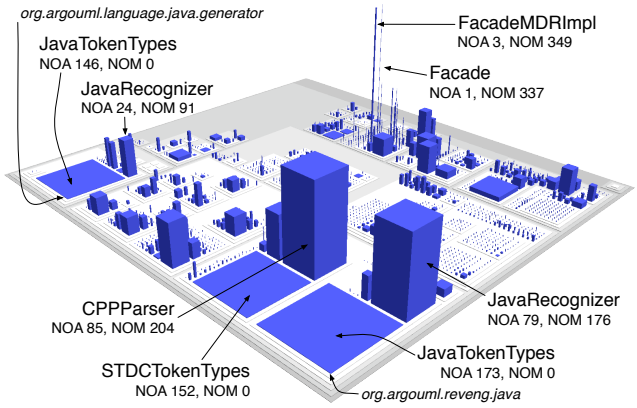

Figure 2. The city of ArgoUML

The city exhibits an interesting symbiosis: parking lots closely-located to massive buildings. After inspecting these entities and their relations, we learned that the three parking lots are interfaces defining many attributes, while the massive buildings are parser classes, which use the attributes (*i.e.,* tokens) defined in these interfaces. Besides the pair for C++ code (*i.e.,*STDCTokenTypes and CPPParser), there are two homonymous pairs for Java code (JavaTokenTypes and JavaRecognizer) defined in different packages. Apart from names, the pairs also share large amounts of duplicated code, which can be refactored. Our hypothesis at the time was that one pair was gradually replacing the other, with both co-existing in the system during the process.

Another striking pair is composed of the two skyscrapers representing the interface Facade and the class FacadeMDRImpl, which dominate the city's top. A closer look reveals an odd situation: there is no other class which implements the large number of methods defined in the Facade interface (*i.e.,* over 300). Whether there were other implementors in the past is again a matter of system evolution.

3.2. Evolution Analysis

The unanswered questions left behind by the previous experiment only strengthened our belief that the history of a software system carries important insights, which cannot be revealed outside the evolutionary context. Therefore, we devised a number of visual techniques for evolution analysis [19]. The one called *time travel* allows stepping through the versions of the system and observing the changes inside the city. To enable such observations we ensure consistent locality, *i.e.,* each artifact representing a software entity is assigned a lifetime real-estate in the city. The empty spaces left behind by the removal of entities are never reallocated.

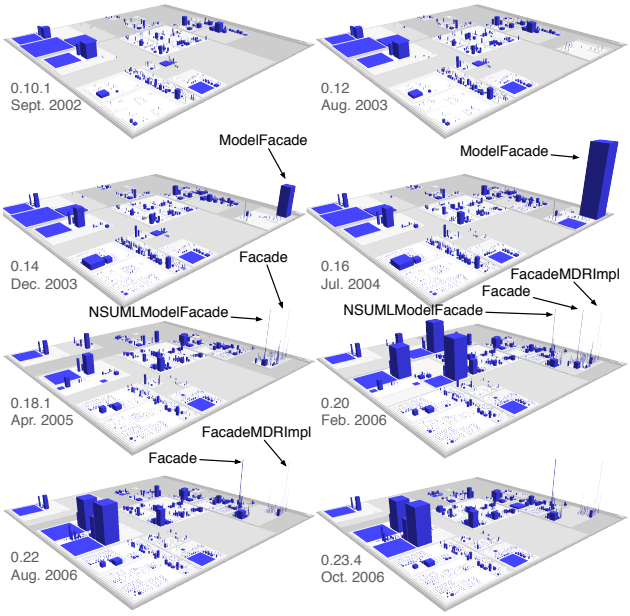

Figure 3. Time travel in ArgoUML's history

Traveling through ArgoUML's history (Figure 3) sheds light on the case of the only implementing class of the Facade interface. In release 0.14 the large ModelFacade class appears, then explodes in size in 0.16. Release 0.18.1 carries the signs of a large refactoring: The removal of ModelFacade coincides with the appearance of an interface (*i.e.,*Facade) and a class (*i.e.,*NSUMLModelFacade) of the same size. Our hypothesis is confirmed by version 0.20, when a second implementing class (*i.e.,*FacadeMDRImpl) appears, justifying the existence of the interface. In release 0.22, the first implementor class is removed, leaving FacadeMDRImpl the only implementor of Facade to these days. A developer of ArgoUML confirmed our hypothesis.

By applying another technique called *age map*, discussed in [19], we learned a fact that discards our other hypothesis: the two pairs called JavaTokenTypes and JavaRecognizer were part of the system from the very beginning.

3.3. Assessing the Quality of the Design

Although software metrics can hint to de sign problems, relying solely on metrics is not accurate enough and often leads to false results. We base our design quality assessment on the results of applying detection strategies [14] to reveal design disharmonies [11]. Our *disharmony map* technique [20] integrates the design problem data in the code cities, which allows us to localize the affected elements and assess the distribution of design disharmonies throughout the system. Inspired by disease maps, we assign vivid colors to the design harmony breakers and shades of gray to the unaffected entities. This enables us to focus on the design disharmonies in the presence of a non-distracting context.

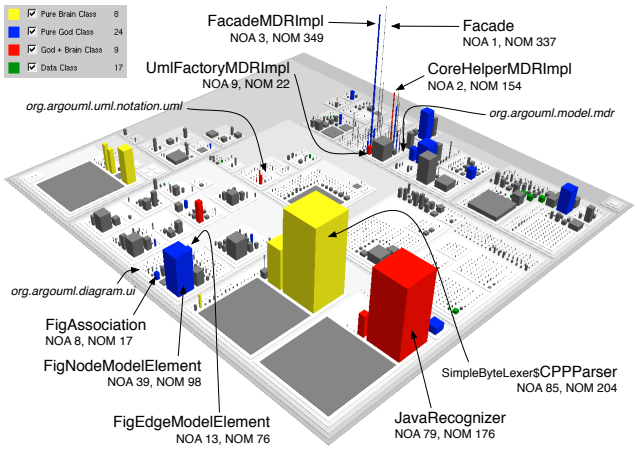

Figure 4. ArgoUML's design problems

ArgoUML has 17 *Brain Classes* (yellow) and 33 *God Classes* (blue), 9 of which are affected by both disharmonies (red), and 17 *Data Classes* (green), distributed rather sparsely throughout the system, as Figure 4 shows. Some of the disharmonious classes are not surprising, given their high number of methods, such as the massive JavaRecognizer and CPPParser, which both happen to be generated classes that do not require manual maintenance. Package org.argouml.model.mdr hosts many problematic classes, including the *God Class* FacadeMDRImpl (3 attributes, 349 methods). A less obvious example are the 3 *God Classes* FigNodeModelElement (39 attributes, 98 methods), FigEdgeModelElement (13 attributes, 76 methods) and FigAssociation (8 attributes, 17 methods), which are core classes of the system and thus subjected to continuous maintenance. A disturbing case appears in package org.argouml.uml.notation.uml with one rather small and three barely visible *God & Brain Classes*: although the four classes have 8 to 24 methods, they contain incredible amounts of code (*i.e.,* 450 to 1,538 LOC), which explains why they are detected as design harmony breakers.

3.4. Experimental Results

During the program comprehension experiment, our approach brought to light a number of interesting cases, which represent potential starting points for upcoming maintenance efforts. However, tracking the origins of a problem requires observing the system's history. The evolutionary visualizations helped us (in)validate our initial hypotheses, before the "reality check" with the system developers. Finally, the disharmony maps pointed out further candidates for refactoring, in the form of actual design problems. Overall, the insights obtained with our complementary visualizations lead to a holistic view of the system.

4. Related Work

3D visualizations have been around for more than a decade [15]. Over the last years, several approaches based on a city metaphor have been proposed. Knight *et al.* [8] and Charter *et al.* [4] use a city metaphor to explore software systems, but at a finer granularity level (*i.e.,* methods are buildings and classes as districts), which does not scale.

Balzer *et al.* propose a very interesting type of 3D visualization, called software landscapes, [2] to visualize single versions of software systems. The drawback of their approach is that it does not visualize system evolution. Moreover, due to their level-of-detail-based navigation, it is not able to produce a "big picture" of the system.

In their approach based on poly cylinders, Marcus *et al.* [13] use the third dimension of this city-like metaphor to map more metrics on the artifacts. The major advantage compared to this approach is the ability of our approach to go beyond single version analysis and to include additional perspectives, such as the disharmony map.

Langelier *et al.* [9] have a similar approach to ours. They use 3D visualizations to display structural information by representing classes as boxes with metrics mapped on height, color and twist, and packages as borders around the classes placed using a tree layout or a sunburst layout. These layouts, while very appealing, do not enable an easy interpretation of the package hierarchy as our layout does. The authors also target design problems, however by visually correlating several metrics to find candidates. By using the results of detection strategies, we depict the real problems in a system, without the risk of getting false positives or false negatives. Recently, the same authors extended their approach to evolution analysis using animations [10]. Unfortunately, the idea is not backed up by a configurable tool and the authors present an analysis of class-level changes only. Our approach allows us to observe changes also at the method level, which led to many intriguing results [19].

Another advantage of our approach over all the presented related work is the availability of the supporting tool.

5. Tool Support

We built CodeCity [18] on top of the Moose framework [6], which provides an implementation of the language-independent FAMIX [5] meta-model for object-oriented systems. FAMIX's language-independence allows us to visualize systems written in several programming languages, including Java, C++, and Smalltalk. Since we first released it in March 2008, CodeCity is freely available[2] and has been downloaded more than 1,700 times over a period of about 11 months.

6. Conclusions and Future Work

We have presented an integrated visual approach including visualization techniques aimed at supporting program comprehension, design quality assessment, and evolution analysis tasks. By using CodeCity, the tool we implemented to support our research, we applied our approach on several open-source systems, represented here by ArgoUML.

While the entity representation, the mapping techniques, and the layouts are in place [16], our approach still lacks a meaningful representation for the relations between entities. Finding efficient ways to express the various relations (*e.g.,* inheritance, invocation, access) is the main direction for our future work. Besides working on explicit representations for relations, we envision devising implicit ones, such as making relations a decisive factor in the layout of the entities (*e.g.,* force-based layouts).

Another possible direction is exploring ways to detect and encode in our code cities higher-level information about the systems, *e.g.,* an architectural view.

Acknowledgments. We gratefully acknowledge the financial support of the Hasler Foundation for the project "EvoSpaces - Multi-dimensional navigation spaces for software evolution" (Hasler Foundation MMI Project No. 1976).

References

[1] S. Alam and P. Dugerdil. Evospaces visualization tool: Exploring software architecture in 3d. In *Proceedings of 14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 269–270. IEEE Computer Society, 2007.

[2] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *VisSym 2004, Symposium on Visualization*, pages 261–266. Eurographics Association, 2004.

[3] S. Boccuzzo and H. C. Gall. Cocoviz: Towards cognitive software visualizations. In *Proceedings of IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*, pages 72–79. IEEE Computer Society, 2007.

[4] S. M. Charters, C. Knight, N. Thomas, and M. Munro. Visualisation for informed decision making; from code to components. In *International Conference on Software Engineering and Knowledge Engineering (SEKE '02)*, pages 765–772. ACM Press, 2002.

[5] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[6] S. Ducasse, T. Gîrba, and O. Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, Sept. 2005.

[7] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[8] C. Knight and M. C. Munro. Virtual but visible software. In *International Conference on Information Visualisation*, pages 198–205, 2000.

[9] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE*, pages 214–223, 2005.

[10] G. Langelier, H. A. Sahraoui, and P. Poulin. Exploring the evolution of software quality with animated visualization. In *IEEE Symposium on Visual Languages and Human-Centric Computing 2008*, pages 13–20. IEEE Computer Society, 2008.

[11] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[12] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[13] A. Marcus, L. Feng, and J. I. Maletic. 3d representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–36. IEEE, 2003.

[14] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 350–359, Los Alamitos CA, 2004. IEEE Computer Society Press.

[15] S. P. Reiss. An engine for the 3d visualization of program information. *Journal of Visual Languages and Computing*, 6(3):299–323, 1995.

[16] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 231–240. IEEE CS Press, 2007.

[17] R. Wettel and M. Lanza. Visualizing software systems as cities. In *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 92–99. IEEE CS Press, 2007.

[18] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering*, pages 921–922. ACM, 2008.

[19] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Proceedings of WCRE 2008 (15th Working Conference on Reverse Engineering)*, pages 219–228. IEEE CS Press, 2008.

[20] R. Wettel and M. Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of Softvis 2008 (4th International ACM Symposium on Software Visualization)*, pages 155–164. ACM Press, 2008.

[21] M. Zelkowitz, A. Shaw, and J. Gannon. *Principles of Software Engineering and Design*. Prentice Hall, 1979.

[2] `http://inf.unisi.ch/phd/wettel/codecity.html`