Visualizing the Evolution of Software Systems using the Forest Metaphor

Ugo Erra, Giuseppe Scanniello, Nicola Capece
Università della Basilicata, Dipartimento di Matematica e Informatica
ugo.erra@unibas.it, giuseppe.scanniello@unibas.it, nicola.capece@unibas.it

Abstract

We present an approach based on a forest metaphor to ease the comprehension of evolving object oriented software systems. The approach takes advantages of familiar concepts such as forests of trees, sub-forest of trees, trunks, branches, leaves, and color of the leaves. In particular, each release of a software is represented as a forest that users (or software maintainers) can navigate and interact with. Users can pass from a release to another one, so understanding how the entire software and its classes evolve throughout the past releases. The approach has been implemented in a prototype of a 3D interactive environment. A preliminary empirical evaluation has been also conducted to assess that environment and the underlying approach.

1 Introduction

In contrast with software development that typically takes 1-2 years, software maintenance lasts for many years after the first release deployed [25]. In fact, a software is changed and evolved for several reasons that range from the correction of faults to the introduction of new functionality [16]. The execution of maintenance operations has the effect of increasing the size and the complexity of the system and gradually decays the original design and the overall quality of the software. The understanding of the evolution of a subject software system is vital in todays software industry [4].

Software visualization is widely employed to comprehend source code and to understand the evolution of a software (e.g., [2], [4], [5]). In particular, a number of metaphors and supporting tools based on 2D and 3D environments have been proposed [6], [11], [15], [21]. Although these techniques are useful to comprehend a subject software and its evolution (e.g., software change [4]), they often fail to show relevant information (e.g., the presence of comment in the code).

In this paper[1], we propose an approach to visualize evolving object oriented software system based on a forest metaphor [7]. In particular, a release of a subject software is visualized as a forest of trees that a user (or software maintainer) navigate and interact with. Visual properties of trees (e.g., trunks and leaves) are mapped according to well defined rules with the metrics extracted from the source code. The maintainer can go also throughout the releases of the subject software to understand its evolution at three different granularity levels: system, package, and class. A 3D interactive environment implementing our approach is proposed as well. To validate this environment and the underling approach, we have also conducted a preliminary case study on three evolving open source software implemented in Java.

[1]Please read the paper on-screen or as a color-printed paper version, we make extensive use of color pictures.

The work presented here is based on [7] and with respect to it the following main contributions are provided: (1) an improved version of the metaphor; (2) the application of the metaphor to software evolution; (3) a extension of the interactive 3D environment; and (4) the results of a case study on 30 releases of three open source software.

The remainder of the paper is organized as follows: in Section 2, we present related work, while we describe the approach in Section 3. In Section 4, we present the preliminary empirical evaluation, while Section 5 concludes the paper presenting future work.

2 Related Work

The city metaphor is one of the most explored natural metaphors for software visualization (e.g., [12], [3], [17], [13]). For example, Wettel and Lanza [22] propose a city metaphor for the the comprehension of object oriented software systems. Classes are represented as buildings and packages as districts. Our approach is different because it is able to visualize more information at low-scale understanding (e.g., number of public methods for each class). Recently, the same authors [23] present a controlled experiment with professionals to assess the validity of both the city metaphor and their 3D interactive environment. The results show that their environment leads to a statistically significant improvement in terms of task correctness and statistically significantly reduces the task completion time. A different metaphor based on a natural environment is presented in [10]. In particular, the authors propose a solar system where each sun represents a package, while planets are classes, and orbits represent the inheritance level within the package. Such metaphor is used to analyze either static or evolving code and to show suspected risk parts of the code. Several are the differences between our proposal and the approaches discussed above. The most remarkable one is that our proposal offers a proper representation for methods, attributes, and source code comment. A possible drawback that affects our metaphor is that it could result complex in case the maintainer is not properly trained. This issue is directly connected to the considerable amount of information our metaphor is able to visually summarize. Another remarkable difference is that all the approaches above do not provide any support to understand the evolution of a software.

In [8] a 3D visual representation for analyzing a software systems release history is proposed. The approach is based on a retrospective analysis technique to evaluate architectural stability, based on the use of colors to depict changes in different releases. Differently, Tu and Godfrey [19] propose an approach that makes an integrated use of software metrics, visualization, and origin analysis. In [9] is suggested an approach based on the notion of history to analyze how changes appear in the software. The authors also propose a tool for visualizing the histories of evolving class hierarchies. The main difference between these approaches and ours is that we consider all the releases of all the source code artifacts, while these approaches consider snapshots of the system. Furthermore, our proposal is based on a natural environment and is able to show both large- and low- scale understanding of evolving software.

3 The Approach

We propose an approach that visualizes object oriented software systems as 3D graving forests of trees. Maintainers can freely navigate and interact with the forests to improve the comprehension of a subject system. Maintainers can also skip from a release of that system to the subsequent or previous ones. The rationale for adopting the forest metaphor and defining our approach are related to large- and low- scale understanding concerns and can be summarized as follows:

- A tree visualization summarizes complex information in a natural way.
- A tree is a complex structure composed of unmistakable elements such as trunk, branches, leaves, and color. These properties can be meaningful mapped onto source code characteristics.
- A forest of trees provides developers with a large-scale understanding of the design and the proliferation of classes.

The Rendering of the Trees. The model upon which we based the creation and the rendering of the trees is based on the Weber and Penn approach [20], which uses a model to create the geometrical structure of trees. This model handles several parameters to modify the property of a tree. Examples are: the height of the tree, the width of the trunk, the level of recursion of the branches, leaves orientation, and so on. To build trees, the approach needs knowledge of neither botany nor complex mathematical principles. For example, each branch may have similarity with its parent and inherit attributes from it. Then, all the branches are influenced by the primary branches. In our proposal, we consider a subset of the parameters needed to render a tree, so affecting its shape (trunk and foliage) according to the main characteristics of the class the tree represents.

Mapping. We considered the following visual properties of a tree: height, branch number, branch direction, leave number, leave color, leaves size, and base size (i.e., the part of trunk without branches). On the other hand, the metrics exploited to influence the visual appearance of a tree are: lines of code (LOCs), lines of comment (CLOCs), number of attributes (NOAs), number of public methods (NEMs), number of private methods (NOMs), and the total number of methods (NEOMs). Although there are many other source code metrics available in the literature, we considered only the most widely known (e.g., [14]). The rationale for this choice relies on the fact that different metrics may complicate the metaphor.

In Table 1, we summarize the mapping between metrics and visual properties of a tree. Some visual properties highlight the local characteristic of a given class (e.g., NOMs/NEOMs), while others make sense only in case trees are analyzed together (e.g., LOCs). In particular, we mapped the height of a tree with LOCs, while the number of branches that sprout from the trunk corresponds to NEOMs. A tree with few branches represents a class with few methods. To highlight the number of public methods, we use branch orientation. In case the value $NEMs/NEOMs$ is close to one, the class has many public methods and its tree has branches pointing out. Differently, if $NEMs/NEOMs$ tends to zero, the class has many private methods and its tree has branches oriented parallel to ground. NOAs is mapped onto the number of leaves, while the leave size is equal to $1/NOAs$. Then, a class with few attributes is represented as a tree with large leaves. A large number of leaves indicates a class with many attributes. The color of the leaves ranges from green to brown. A class having the number of CLOCs greater than LOCs is represented as a tree with a green foliage, brown otherwise. Finally, the base size of a tree is computed as $NOMs/NEOMs$. A value close to one indicates that a class

METRICS	VISUAL PROPERTIES
LOCs	Tall
NEOMs	Branches
NOAs	Number of leaves
NEMs/NEOMs	Branches orientation
1/NOAs	Size of leaves
LOCs/CLOCs	Color of the leaves (from green to brown)
NOMs/NEOMs	Base size
LOCs = Lines of Code	
CLOCs = Lines of Comment	
NOAs = Number of Attributes	
NEOMs = Total Number of Methods	
NEMs = Number of Public Methods	
NOMs = Nmber of Private Methods	

Table 1: Metrics and visual properties mapping.

has many private methods and then the tree has a large base size. Conversely, if NOMs/NEOMs is close to zero, the class has many public methods and its tree has a small base size. The mappings between the properties of a tree and the selected metrics try as much as possible to ease the comprehension of systems, packages, and classes [7].

Sample Trees. Figure 1 shows the trees of three sample classes for the systems considered in the case study (i.e., the main classes for JFreeChart and JEdit, respectively, and the class for JSheet). The trees of two subsequent releases of these classes are shown as well. As far as the JFreeChart class is concerned, we can note that passing from the release 0.5.6 to the 0.7.0 the foliage is different. In particular, the number of the leaves is larger in the last release and the color of these leaves is greener. This indicates that the number of attributes increased. Moreover, the class in the last release is better commented. For the class selected in the JEdit system, we can observe a different shape of the threes. This difference is due to the total number of methods that increased in the class of the release 4.3. The different relationships between public and private methods also affected the shapes of the trees: the number of public methods is much larger than those private. Regarding the class considered for JHotDraw, we can observe that the branches of all the trees point out, so indicating that the number of public methods is close to the number of the private ones. The main difference among the three trees is that the total number of methods is larger in the releases 7.0.8 and 7.1. This difference results in a slight difference in the tree shapes.

Regarding the software evolution, if a class appears in many releases of the same software system, it will be always shown in the same point on the ground of the forests corresponding to these releases. The position is determined considering all the classes present in all the releases of the system and using a spiral pattern. Therefore, if a class is present in a release of the subject system the corresponding tree is shown, otherwise it is not shown. Each spiral represents a package of a subject system. Only for space reasons, we do not provide here a forest visualization for the different releases of the software systems considered.

Implementation. The proposed approach has been implemented in a prototype of a supporting system. It is composed of three main components. The former extracts all the metrics needed for the visualization and produces an XML file. This design choice allows making independent the extraction of the measures from the rendering engine. This component is implemented in Java. The second component maps the tree parameters and the extracted metrics. Also this component is implemented in Java and produces an XML file that is then used for the rendering of all the forest. To implement the 3D environment, we used the OpenTree library [1]. This library provides 3D tree generation for real time applications. OpenTree is a cross-platform and engine-independent library written in C++. The library uses an array of vertex to generate mesh data to render trees. Vertex can be used by any graphics library. We used here the OpenGL [24] graphics library.

The tool supports maintainers in the exploration of the 3D forests using the Visual Information Seeking Mantra Overview, zoom and filter, and details-on-demand [18]:

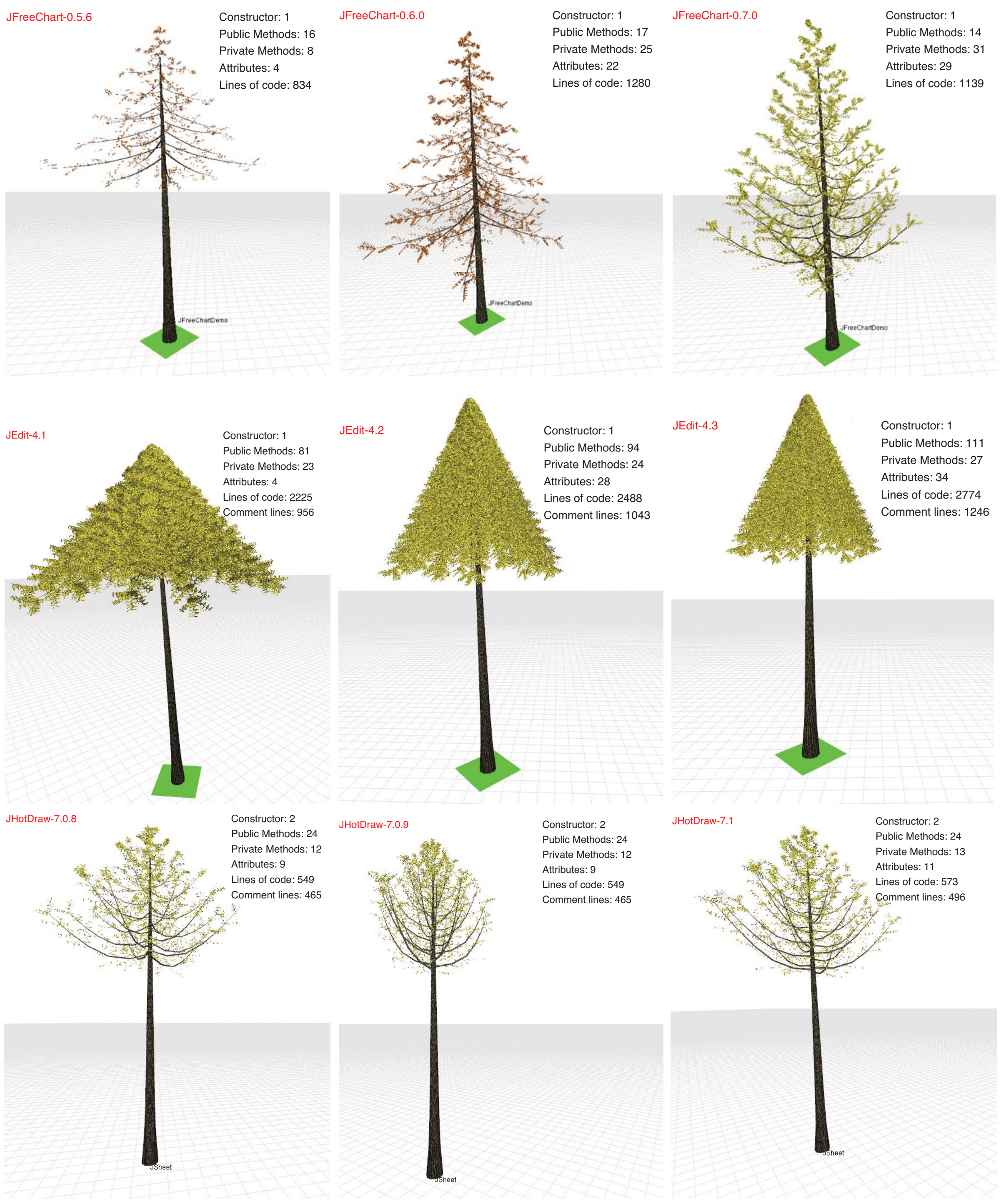
- *Overview.* The 3D environment supports the visualization of the entire forest. The overview provides the user with an interactive visualization of all the classes of a subject system by using mouse and keyboard. The name of each class is attached as a label to the corresponding tree. These labels can be hidden if needed.

- *Zoom and filter.* The environment allows the user to zoom in from the initial overview refining the current view. If the maintainer identifies a tree of interest in the overview, this tree can be selected and explored. The zoom and filter feature is particularly useful in case of large software.

- *Details-on-demand.* The maintainer can select one tree and show their properties. Each parameter is labeled with the class membership and the tree information (i.e., the values for all the metric used in the tree rendering) is shown if request.

4 Evaluation

To validate our 3D environment and the underlying approach, we conducted a preliminarily case study on 30 releases of three open source object oriented software systems implemented in Java:



JFreeChart-0.5.6
Constructor: 1
Public Methods: 16
Private Methods: 8
Attributes: 4
Lines of code: 834

JFreeChart-0.6.0
Constructor: 1
Public Methods: 17
Private Methods: 25
Attributes: 22
Lines of code: 1280

JFreeChart-0.7.0
Constructor: 1
Public Methods: 14
Private Methods: 31
Attributes: 29
Lines of code: 1139

JEdit-4.1
Constructor: 1
Public Methods: 81
Private Methods: 23
Attributes: 4
Lines of code: 2225
Comment lines: 956

JEdit-4.2
Constructor: 1
Public Methods: 94
Private Methods: 24
Attributes: 28
Lines of code: 2488
Comment lines: 1043

JEdit-4.3
Constructor: 1
Public Methods: 111
Private Methods: 27
Attributes: 34
Lines of code: 2774
Comment lines: 1246

JHotDraw-7.0.8
Constructor: 2
Public Methods: 24
Private Methods: 12
Attributes: 9
Lines of code: 549
Comment lines: 465

JHotDraw-7.0.9
Constructor: 2
Public Methods: 24
Private Methods: 12
Attributes: 9
Lines of code: 549
Comment lines: 465

JHotDraw-7.1
Constructor: 2
Public Methods: 24
Private Methods: 13
Attributes: 11
Lines of code: 573
Comment lines: 496

Figure 1: Three evolving classes for the systems JFreeChart, JEdit, and JSheet.

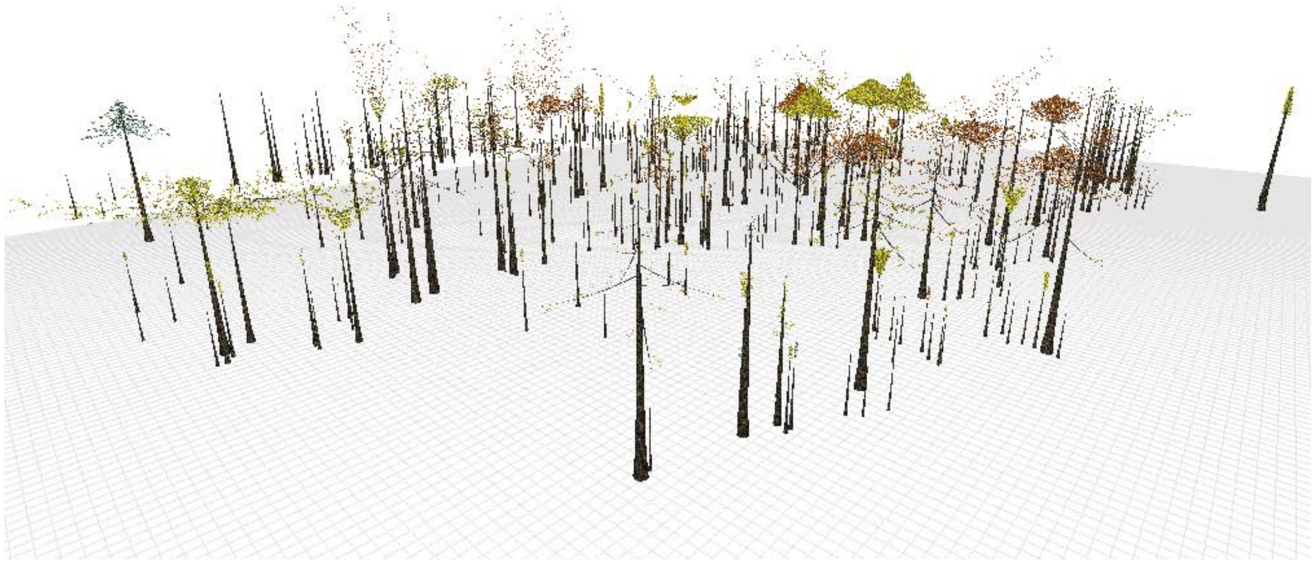


Figure 2: The forest for JEdit ver. 4.3

- **JFreeChart**[2] is a chart library that makes it easy for developers to display charts in their applications;
- **JEdit**[3] is a programmer's text editor with an extensible plug-in architectures;
- **JHotDraw**[4] is a graphical user interface framework for technical and structured graphics;

We considered 6 releases for JFreeChart (from 0.5.6 to 0.7.3), 16 for JEdit (from 3.0 to 4.3.2), and 8 for JHotDraw (from 7.0.7 to 7.4.1).

We selected these systems: *(i)* to verify whether the validity of our proposal is affected by the kind of system used; *(ii)* because they have been widely used in the past to assess the validity of tools for supporting maintenance tasks; and *(iii)* because they present some differences in the feature implemented.

Figure 2 shows the forest of JEdit 4.2. We can observe that there are some classes that contained a high number of lines of comments. Among these classes, there are some classes with a larger number of public methods and with a few attributes. On the other hand, there are few classes with a large number of attributes and a few number of public methods. In the first case, the classes provide services to other classes and to make easy how to use these services the developers commented the code. In the latter case, the trees represent property classes. The analysis of the 16 releases of JEdit clearly shows that the size of the system increased throughout the releases analyzed. In particular, both the number of classes and packages increased. The big picture of the forests did not show particular evolution patterns. Differently, on JFreeChart we observed that the color of the foliage became greener and greener throughout its releases. This result indicates that developers paid more attention in commenting the code in the last analyzed releases.

[2]www.jfree.org/jfreechart
[3]www.jedit.org
[4]www.jhotdraw.org

Further Results. The use of our 3D environment on the selected systems leads also to the following two considerations: *(i) Scalability*. The environment did not scale up very well on large software systems and when the number of the releases to be analyzed increase (e.g., JEdit). To address this issue, we plan to implement a new version of our environment based on a state-of-the-art 3D engine. *(ii) Completeness*. Our approach and environment provide a fair amount of information for an overview of the system and also offer a proper representation for methods, attributes, and comments.

5 Conclusion and Future Work

In this paper, we presented an approach for the visualization of evolving software systems. The approach is based on the forest metaphor presented in [7]. The approach has been implemented in prototype of a supporting tool. It provides features to navigate in the forest as a free fly 3D virtual camera. Our prototype also implements zoom features and visualizes several attributes of the classes (e.g., the names and the values of the metrics) to promote the comprehension at fine-grained level.

To assess the validity of the approach, we have also conducted a preliminary case study on three open source software systems implemented in Java. The achieved results seem encouraging, but due to the preliminary nature of our investigation caution is needed. To increase our awareness on the achieved findings, we plan to conduct further empirical investigations on different releases of evolving software systems. We also plan to conduct users' studies to evaluate the effectiveness of our proposal in the execution of maintenance or comprehension tasks. Finally, future work will be devoted to increase the realism of the forest and to assess whether users perform maintenance tasks in a more efficient fashion in case of more realistic forests.

References

[1] OpenTree Library. `http://opentreelib.sourceforge.net/`.

[2] Software visualization: Programming as a multimedia experience, 1998.

[3] S. M. Charters, C. Knight, N. Thomas, and M. Munro. Visualisation for informed decision making; from code to components. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 765–772, New York, NY, USA, 2002. ACM.

[4] M. D'Ambros and M. Lanza. Visual software evolution reconstruction. *Journal of Software Maintenance*, 21(3):217–232, 2009.

[5] S. Diehl, editor. *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*, volume 2269 of *Lecture Notes in Computer Science*. Springer, 2002.

[6] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Trans. Software Eng.*, 31(1):75–90, 2005.

[7] U. Erra and G. Scanniello. Towards the visualization of software systems as 3d forests: the codetrees environment. In *ACM SAC*, page to appear, 2012.

[8] H. Gall and M. Jazayeri. Visualizing software release histories: The use of color and third dimension. pages 99–108. IEEE Computer Society Press, 1999.

[9] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *In Proceedings of ICSM 04 (International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 2004.

[10] H. Graham, H. Y. Yang, and R. Berrigan. A solar system metaphor for 3d visualisation of object oriented software metrics. In *Proceedings of the 2004 Australasian symposium on Information Visualisation - Volume 35*, APVis '04, pages 53–59, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[11] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu. Software evolution comprehension: Replay to the rescue. In *ICPC*, pages 161–170, 2011.

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

[13] B. Kot, B. Wuensche, J. Grundy, and J. Hosking. Information visualisation utilising 3d computer game engines case study: a source code comprehension tool. In *Proceedings of the 6th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction: making CHI natural*, CHINZ '05, pages 53–60, New York, NY, USA, 2005. ACM.

[14] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems*. Springer Verlag, 2010.

[15] A. Marcus, L. Feng, and J. I. Maletic. Comprehension of software analysis data using 3d visualization. In *IWPC*, pages 105–114, 2003.

[16] M.M. and Lehman. Program evolution. *Information Processing & Management*, 20(12):19 – 36, 1984. ¡ce:title¿Special Issue Empirical Foundations of Information and Software Science¡/ce:title¿.

[17] T. Panas, R. Berrigan, and J. Grundy. A 3d metaphor for software production visualization. In *Proceedings of the Seventh International Conference on Information Visualization*, pages 314–, Washington, DC, USA, 2003. IEEE Computer Society.

[18] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society.

[19] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *In 10th International Workshop on Program Comprehension (IWPC02*, pages 127–136. IEEE Computer Society Press, 2002.

[20] J. Weber and J. Penn. Creation and rendering of realistic trees. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 119–128, New York, NY, USA, 1995. ACM.

[21] R. Wettel and M. Lanza. Program comprehension through software habitability. In *ICPC*, pages 231–240, 2007.

[22] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, pages 921–922, New York, NY, USA, 2008. ACM.

[23] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceedings of ICSE 2011 (33rd International Conference on Software Engineering)*, page to be published, 2011.

[24] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[25] M. V. Zelkowitz, A. C. Shaw, and J. D. Gannon. Principles of software engineering and design. 1979.