Sensorial software evolution comprehension

Subtitle: Reinventing the World

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Dependable Distributed Systems

presented by
Gianlorenzo Occhipinti

under the supervision of
Prof. Michele Lanza
co-supervised by
Prof. Csaba Nagy, Prof. Csaba Nagy

July 2022

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Gianlorenzo Occhipinti
Lugano, Yesterday July 2022

To my beloved

Someone said . . .

Someone

Abstract

The comprehension of software evolution is essential for the understandability and maintainability of systems. However, the sheer quantity and complexity of the information generated during systems development make the comprehension process challenging. We present an approach, based on the concept of synesthesia (the production of a sense impression relating to one sense by stimulation of another sense), which represents the evolutionary process through an interactive visual depiction of the evolving software artifacts complemented by an auditive portrayal of the evolution. The approach is exemplified in SYN, a web application, which enables sensorial software evolution comprehension. We applied SYN on real-life systems and presented several insights and reflections.

Acknowledgements

ACK

Contents

Figures

Tables

Chapter 1

Introduction

In 1971 Dijkstra, made an analogy between computer programming an art. It stated that is not important to learn how to compose a software, but instead, it is important to develop its own style and what will be their implications.

Software Understanding

- Section [Challenges of software understanding] - Section [Challenges of software maintainability] - Section [Software comprehension] - Section [Our approach] We present an approach, based on the concept of synesthesia (the production of a sense impression relating to one sense by stimulation of another sense), which represents the evolutionary process through an interactive visual depiction of the evolving software artifacts complemented by an auditive portrayal of the evolution.

Chapter 2

State of the art

2.1 Software visualization

Essential parts of software lifecycles are software maintenance and software evolution. Both activities require the comprehension of the system by the developer. Mayrhauser [13] defined program comprehension as a process that uses knowledge to acquire new knowledge. Generally, programmers possess two types of knowledge: general knowledge and software-specific knowledge, which represent their level of understanding of that software. Software comprehension aims to increase this specific knowledge of the system, and, to do that, it can leverage some software visualization techniques. Software visualization supports the understanding of software systems because it enables the visualization of the system's information (architecture, source code, behavior) with a 2D or 3D representation. Stasko et al.[5] conducted a study in 1998 that shows how visualization arguments human memory since it works as external cognitive aid and thus, improves thinking and analysis capabilities.

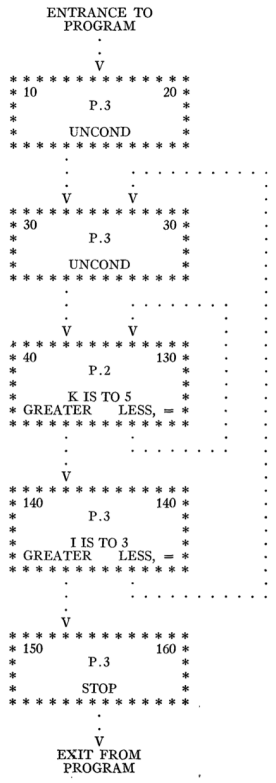


Figure 2.1. Flowchart presented by Haibt in 1959

History of software visualization

The earliest form of software visualization found in the literature was in the form of 2D diagrams. Haibt [6] in 1959, was one of the first to use them to visualize software systems. He provided a graphical outline of a program and its behavior with flowcharts. As shown in Figure 2.1, they were 2D diagrams that described the execution of a program. He wrapped each statement in a box, representing the control flow with arrows. Ten years later, the effectiveness of flowcharts was confirmed by Knuth [9]. Un-

fortunately, at that time, most of the programs were affected by
a lack of readability since they were not well documented. So, it introduced a tool to automatically generate visualizations from the software documentation. Nassi and Schneiderman[12], in 1973, introduced the Nassi–Shneiderman diagram (NSD), able to represent the structure of a program. The diagram was divided into multiple sub-block, each with a given semantic based on its shape and position.

The 80s registered two main directions of software visualization. The first was the source code presentation. Hueras [7], and Waters [14] developed two techniques to format the source code with a prettyprinter. The second direction was the program behavior, used mainly for educational purposes. One of the most prominent visualization systems of that period was Balsa-II [1]. Around the end of the 80s, Müller et al. [11] released Rigi, a tool able to visualize large programs. It exploited graph model, argumented with an abstraction mechanisms, to represent systems components and their relationships.

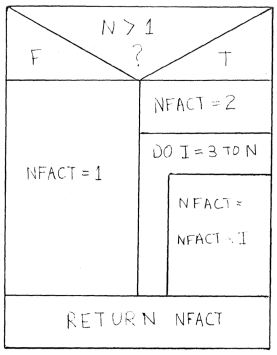


Figure 2.2. NSD of the factorial function.

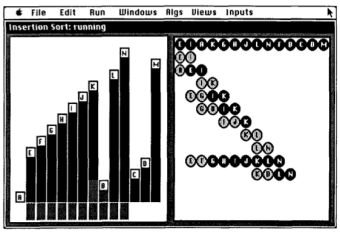


Figure 2.3. Balsa-II

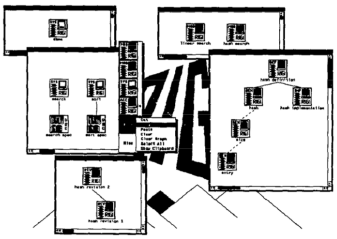


Figure 2.4. Rigi

The 1990s recorded more interest in the field of software visualization. In 1992 Erik et al [4] introduced a new technique to visualize line oriented statistics. It was enboded in Seesoft, a software visualization system that allowed to analyze and visualize up to 50,000 lines of code simultaneously. On their visualization, each line was mapped to a thin row. Each row was associated with a color that described a statistic of interest. One year later, De pauw et al. [3] introduced Jinsight, a tool able to provide animated views of the behavior of object-oriented systems.

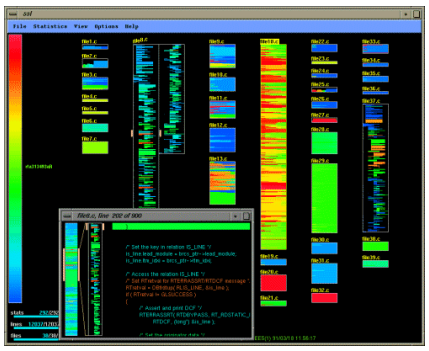


Figure 2.5. Seesoft

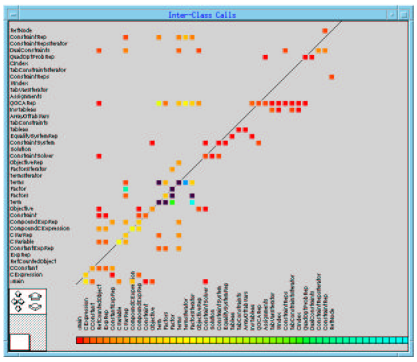


Figure 2.6. Jinsight

That period was favorable also for experimenting novel research directions for visualization, such as 3D visualization and Virtal Reality. In 1998, Chuah and Erick [2] proposed three different techniques to visualize project data. They exploited the concept of glyphs, a graphical object that represents data through visual parameters. The first technique was the Timewhell glyph, used to visualize time-oriented information (number of lines of code, number of errors, number of added lines). The second technique was the 3D wheel glyph; it encoded the same attributes of the time wheel, and additionally, it used the height to encode time. Infobug glyph was the last technique, where each glyph was composed of four parts, each representing essential data of the system (time, code size, number of lines of code added/deleted/modified).

Figure 2.7. Timewhell

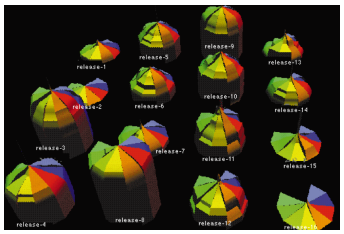


Figure 2.8. 3D wheel

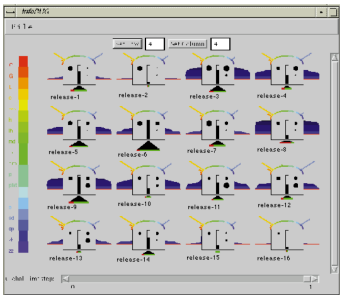


Figure 2.9. Infobug

In the same year, Young and Munro [16] explored representations of software for program comprehension in VR. Finally in 1999, Jacobson et al. [8] introduced what we now know as de facto the standard language to visualize the design on a system: UML.

2.1.1 Software evolution visualization

Before the beginning of the 21st century, visualize the evolution of a system was an unfeasible task due to the lack of data. However, tanks to the spread of version control systems and of the open-souce movement, these information became publicy accessible. As a result, may researchers focused their work on software evolution visualization.

In 2001 Lanza [10] introduced the concept of the Evolution Matrix. It was a way to visualize the evolution of software without dealing with a large amount of complex data. Furthermore, that approach was agnostic to any particular programming language. The Evolution Matrix aimed to display the evolution of classes in object-oriented software systems. Each column represented a version of the software, and each row represented a different version of the same class. The cells were filled with boxes whose size depended on two different metrics. Thanks to this approach, he was able to infer some evolution information by just looking at the shape of the matrix

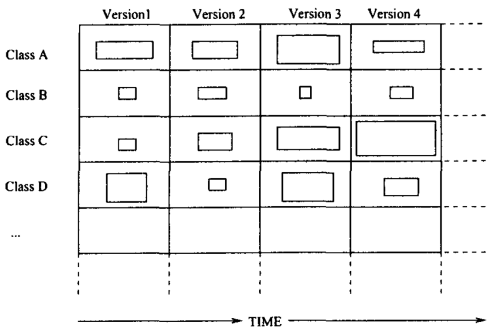


Figure 2.10. A schematic display of the Evolution Matrix

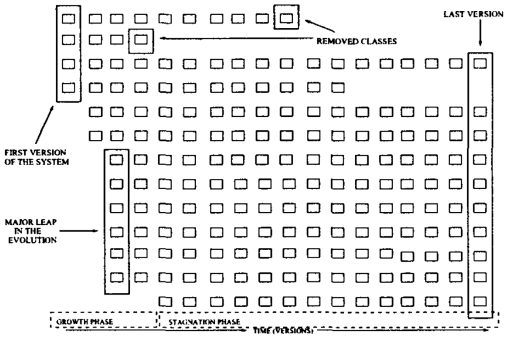
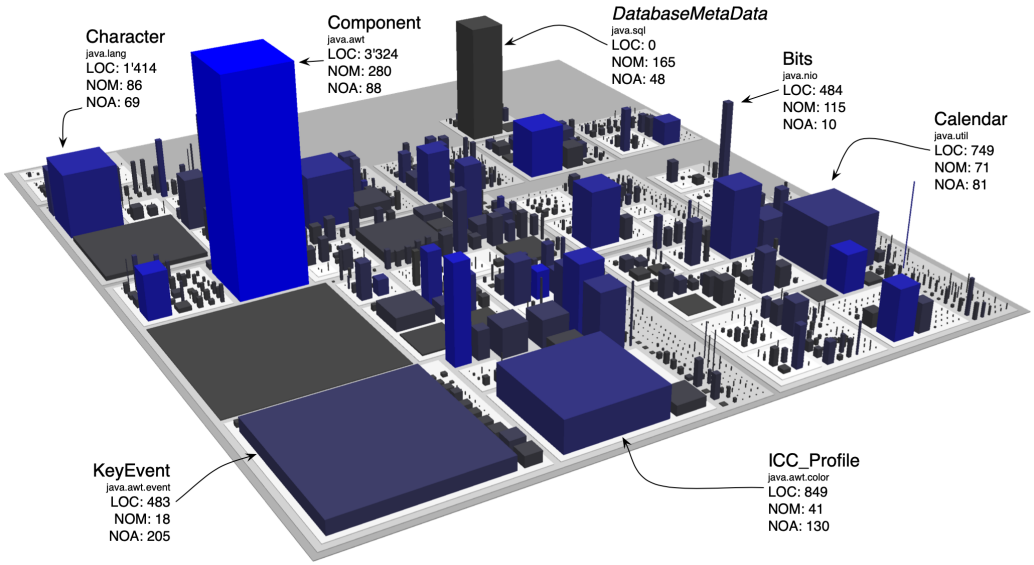


Figure 2.11. Some characteristics of the Evolution Matrix

Wettel, in his thesis [15], defined a city metaphor for software visualization that represents software as cities. His work represents packages as districts and classes as buildings. This metaphor was applied in different contexts related to reverse engineering (program comprehension, software evolution, software quality) to demonstrate metaphor's versatility. As a result, he found evidence that his approach works. However, he claims that city metaphor brings visual and layout limitations (not all visualization techniques fit well with it). Under those circumstances, he preferred simplicity over the accuracy, so he obtained a simple visual language that facilitates comprehension of data. He conducted an experiment of the evidence that the city metaphor enables the creation of efficient software visualizations. His approach was implemented by a software visualization tool called CodeCity that supports the city metaphor.



Bibliography

[1] Marc H. Brown. Exploring algorithms using balsa-ii. *Computer*, 21(5):14–36, may 1988. ISSN 0018-9162. doi: 10.1109/2.56. URL https://doi.org/10.1109/2.56.

[2] M.C. Chuah and S.G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, 1998. doi: 10.1109/38.689658.

[3] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, page 326–337, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915879. doi: 10.1145/165854.165919. URL https://doi.org/10.1145/165854.165919.

[4] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, nov 1992. ISSN 0098-5589. doi: 10.1109/32.177365. URL https://doi.org/10.1109/32.177365.

[5] Jean-Daniel Fekete, Jarke van Wijk, John Stasko, and Chris North. *The Value of Information Visualization*, volume 4950, pages 1–18. 07 2008. ISBN 978-3-540-70955-8. doi: 10.1007/978-3-540-70956-5_1.

[6] Lois M. Haibt. A program to draw multilevel flow charts. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), page 131–137, New York, NY, USA, 1959. Association for Computing Machinery. ISBN 9781450378659. doi: 10.1145/1457838.1457861. URL https://doi.org/10.1145/1457838.1457861.

[7] Jon Hueras and Henry Ledgard. An automatic formatting program for pascal. *SIGPLAN Not.*, 12(7):82–84, jul 1977. ISSN 0362-1340. doi: 10.1145/954639.954645. URL https://doi.org/10.1145/954639.954645.

[8] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN 0201571692.

[9] Donald E. Knuth. Computer-drawn flowcharts. *Commun. ACM*, 6(9):555–563, sep 1963. ISSN 0001-0782. doi: 10.1145/367593.367620. URL https://doi.org/10.1145/367593.367620.

[10] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, page 37–42, New York, NY, USA, 2001. Association

for Computing Machinery. ISBN 1581135084. doi: 10.1145/602461.602467. URL https://doi.org/10.1145/602461.602467.

[11] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, page 80–86, Washington, DC, USA, 1988. IEEE Computer Society Press. ISBN 0897912586.

[12] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Not.*, 8(8):12–26, aug 1973. ISSN 0362-1340. doi: 10.1145/953349.953350. URL https://doi.org/10.1145/953349.953350.

[13] A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995. doi: 10.1109/2.402076.

[14] Richard C. Waters. User format control in a lisp prettyprinter. *ACM Trans. Program. Lang. Syst.*, 5(4):513–531, oct 1983. ISSN 0164-0925. doi: 10.1145/69575.357225. URL https://doi.org/10.1145/69575.357225.

[15] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 551–560, 2011. doi: 10.1145/1985793.1985868.

[16] P. Young and M. Munro. Visualising software in virtual reality. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pages 19–26, 1998. doi: 10.1109/WPC.1998.693276.