To Enlighten Hidden Facts in The Code: A Review of Software Visualization Metaphors

Yangyang XU*, Yan LIU† and Jiabin ZHENG‡
School of Software Engineering, Tongji University
Shanghai, China
Email: *1334902@tongji.edu.cn, †yanliu.sse@tongji.edu.cn, ‡1434321@tongji.edu.cn

***Abstract*—Software visualization has been adopted to help engineers understand the design and functionality better and faster. A number of visualization techniques have been developed in the field of structure, behavior and evolution recently. However, there is little attempt to comprehensively review current state of the art for software professionals. As a consequence, this paper employs a systematic review of research literature on the visualization of code, to identify current application tasks, discuss variety effectiveness of visual representations, and sort out their relationships to improve usability. Finally, unsolved issues and future research opportunities have been discussed.**

Keywords-code visualization; metaphor; mapping; systematic literature review;

I. Introduction

Software systems have increasingly grown in size and complexity. Due to the high turnover rate and changing industry environments, engineers have encountered challenge in software comprehension and maintenance. Particularly, studies indicate that 80% of the software costs are used for maintenance, in which 40% is devoted to understand source code[1]. It is recognized that people are better at deducing information from graphical image than numerical and textual formats[2]. Therefore, Software visualization(SV) is defined as a technique which transfers hidden facts into visual forms like images, diagrams or animations[3]. However, most of the current research focuses on specific technique of analysis, there is little work on how visualization techniques really facilitate general tasks for stakeholders. Critically, the challenge is to find a good visual structure which maintains fulfilled information of tasks and can be perceived easily. As a consequence, SV techniques have not been widely adopted in the industry[4].

The body of work on SV is such that, at first, it is important to identify visualization tasks and select adaptive metaphors. This is because SV should always be goal-oriented[5]. Namely, visualization goals drive the definition of SV techniques. Efforts on this research can be categorized in two approaches: empirical study and literature review. With the limitation of quantitative analysis for SV, we decided to conduct a systematic literature review(SLR) of software code visualization. On the basis of running through state of the art, we could summarize existing visual metaphors, understand analytic tasks and obtain a better mapping between them. The results are expected to be utilized as a foundation for potential experimentation and professional scholars.

DOI reference number: 10.18293/SEKE2015-203

TABLE I. Research questions of SLR

Number	Research question
RQ1	What analytic tasks does current SV support?
RQ2	What types of perspectives do engineers use SV techniques?
RQ3	What types of visual metaphors are available in the study?
RQ4	Which tools are used to support software code visualization?
RQ5	What is the correlation between tasks and visual metaphors?

II. Research methods

A. Planning the Review

Prior to undertaking a SLR it is necessary to identify the purpose[6], namely, to explore relevant literature through research question "how visualization metaphors facilitate tasks supported in current techniques?"

1) Research questions: Relavant research questions(RQs) to guide SLR have been formulated in Table I.

The motivation of RQ1 and RQ2 is to identify the goal of SV techniques. RQ3 and RQ4 get a comprehensive set of available visualization techniques. The objective of RQ5 is to investigate relationship between techniques and supported tasks.

2) Review protocols: With the definition of RQs, it is essential to specify review protocols to reduce possibility of bias[6]. It includes search terms, search resources, selection criteria and data extraction strategy.

In this SLR, initial search terms were "software visualization" and "visualization techniques". Moreover, "visual", "visualize" and other synonyms could be considered as search terms. Query of this review were built mainly in 6 top publication venues of SV area, along with research in 4 representative databases: IEEE Xplore, Sciencedirect, ACM Digital Library and Springer Link.

Selection criteria are listed in Table II. Data items to extract related information are defined as follows: analytic

TABLE II. Inclusion and exclusion criteria of SLR

	Inclusion criteria
1	A study is published after 2007.
2	A study discusses about SV supported tasks, metaphors, tools, and evaluation.
	Exclusion criteria
1	A study does not include code visualization.
2	A study is with little evidence or outdated.
3	A study is duplicate.

TABLE III. Number of paper per step for per venue

Public venue	Search	Selection1	Selection2
ICSE	115	89	12
ICPC	79	70	10
ICSM	77	48	5
SoftVis	96	14	2
WCRE	68	50	7
WICSA	9	8	1

tasks/visualization activities for RQ1, RQ2; visualization representation/metaphors for RQ3; tool support for RQ4; and conclusion/relationship/correlation for RQ5.

B. Conduct the Review

SV papers have been published in many venues. We selected 6 representative ones as paper sources, including ICSE, ICPC, ICSM, SoftVis, WCRE and WICSA. Table III indicates selected number of papers in each step for each venue. Initial results were achieved with query of search terms. Due to the large number of papers, we applied selection criteria in Table II to concentrate the results. We limited the date of publication to consult mature theory in the first selection. And manual search method was performed in the second round. As a result, 37 papers were selected from these venues. In addition, same research method has been explored in 4 famous public databases and we retrieved 50 results. After removing duplicate ones within two approaches, finally we identified 81 papers for the review. Due to the limited space, selected papers are listed in "http://SSE.tongji.edu.cn/liuyan/sv_papers.html".

III. Visualization Review Result

After conducting the review, this section reports results based on the synthesis and analysis of data extraction activities to answer RQs.

A. Supported Tasks

Software visualization focuses on diverse aspects through development stages[7]. Selected papers have indicated an increasing interest in not only visualization of software components, their properties, relationships, but also their evolution, behavior and instruction execution[1][3][7]. In order to present various tasks clearly, this review adopted the classification proposed by Stephan Diehl[3], concerning with visualizing static analysis, dynamic execution of program and evolution of code.

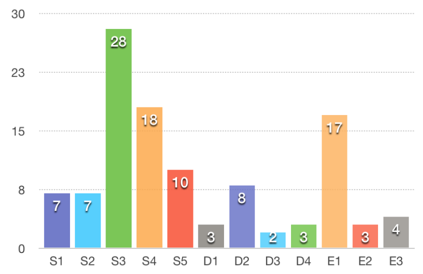

Figure 1. Number of studies per category

1) Static analysis: According to statistics, 54 studies in our selection have reported this aspect, including control-flow analysis[3], code map[8], dependency relationship between software components[9], software architecture[9] and code metrics[7].

Control-flow analysis (S1): It is used to depict sequential order of the program in source code which helps developers to think in an orderly manner[10].

Code map (S2): This visualization maps the relationships between pieces of code. Recent studies have indicated full interests in this analytic task.

Dependency relationship (S3): This is an essential part of software visualization owing to tremendous amount of interactions between components. It provides a visual approach for engineers to obtain an overview of dependencies for entire solution without viewing all the files and lines of source code.

Software architecture (S4): Because the focal point is code visualization, here software architecture focuses on hierarchy. It is one of core topic in SV that is used to help engineers organize artifacts into logical, abstract groups and make sure code remains consistent with the design.

Code metrics (S5): Typical static metrics of source code include size of code, number of components and complexity. However, visualization techniques discussed in 10 studies have no great changes in the past few years.

2) Dynamic analysis: In contrast with static properties of source code, merely 15 studies involved in research support dynamic analysis visualization. The motivation is to present what happens at run time, concerning executed time, statement coverage, dynamic architecture and program slice.

Executed time (D1) and statement coverage (D2) are aimed to optimize system performance[3][11]. 8 studies (R9,R12,R15,R19,R25,R52,R53,R64) mentioned visualization of code coverage which helps users pay more attention on frequent lines and non-executed ones.

Dynamic architecture (D3): Behavior diagrams are generated to describe changes at the level of architecture.

Program slice (D4): A dynamic slice is the set of all program points that actually affect a program point for a given input(R7,R11,R13). This is intended to find patterns in source code, and users can eliminate and sort procedures based on whether or not they are in the slice.

3) Evolution: Tracking changes between different versions can be meaningful for code management and maintenance[12].

Evolution metrics (E1) in this aspect include who edited specific parts of code; when each line was last modified; where bugs were located, who fixed these bugs and how the evolution processed. As a consequence, the visualization can be used in the field of code discovery and code decay[13]. Visualization of software archives (E2) which comes up from a whole overview of system updating has been reported in 3 papers. Added lines, deleted lines and changed lines reflect which class is added or removed in one file version. At the same time, visualization techniques to depict structural changes (E3) are limited[7][13].

Above all, Fig. 1 reveals that majority of selected studies investigate visualization of static aspects especially in dependency relationships and software architecture. While dynamic analysis visualization is a broad and relatively young research field due to the limitations of implementation techniques. Visualization of evolution has attracted great interests in recent year. Nevertheless, most of the analyzed work intends to visualize evolution metrics with fewer on complex structural change.

B. *Principal Visual Representation*

The focal step of the visualization process is to choose effective visual representations for facts in source code. They are built from points, lines, areas, and volumes with various properties: size, length, width, height, volume, position, orientation, angle, slope, color, grayscale, texture, and shape[3].

To better address RQ3, we have tried to combine representations which use different terms with same essence. Those rarely used or supported with little evidence have been excluded. Therefore, Fig. 2 presents various types of visual representations that are currently used in code visualization, ranging from simple to complex.

1) Line representation: Relevant visual representations include pie and bar graph, histograms and pixel representation. It makes the entire file visible with the attributes of length, indentation and color. According to the effective technique of color-coding which is beneficial for layering information, it is possible to show a million lines of code in a screen and make it easy to find different parts. However, it might be shrunk to a single row of pixels which is less readable with large scale programs.

2) Node-link layout: This is the most well-known metaphor to represent the relationship and hierarchy of software components[1]. It uses nodes and links to represent elements(files, packages, classes) and structural relationships respectively. Studies in selected research have indicated that representation becomes too large due to the high interconnectivity between components[7]. Related terms such as Sunburst tree layout and hyperbolic tree layout[14] have moved to more sophisticated ones to deal with the problem.

3) Matrix views: It is an effective visualization to display two-dimensional grid with rows corresponding to one index and columns to another[15]. In contrast to graph-based visualization, this representation provides complementary information for large programs with no overplotting. This is owing to strength of matrix that a single image contains thousands of cells[15]. One weakness of the representation is that adjacent modules are unordered and the display is irrelevant to the structure of source code tree.

4) Treemap: The metaphor is an effective means to visualize hierarchical decompositions of software. It executes tiling algorithm to slice the view into several parts corresponding to the number of subsystem. The space-filling technique visualizes methods as elementary boxes and classes as composed boxes which helps to address space problem in comparison with node-link graph. However, it has common problem with matrix views which is impossible to map the structure.

5) Notation views: This type of visualization presents the relationship between elements in a structure which is currently used for UML diragram. In contrast to node-link layout, it is reported that type of the nodes is important information in notation views.

6) Cityscape views: 15 studies reported this type of visualization which uses physical contexts to represent software components and relationships[16]. Similiar metaphors include forest metaphor (R39) and network view. Comparing with matrix views, it provides more intuitive view for users[17] and enhances the visualization of metrics. Nevertheless, weaknesses of 3D technology like object occlusion and performance issues have limited the usability.

7) Metric views: The implementation of metric view displays information on the top of UML diagram. UML is an visual modeling language to specify, design and construct software systems[18]. An extension metaphor of metric views is "areas of interest" proposed by Byelas and Telea[19].

Visualization representations proposed in research range from simple "line representation" to complex n-dimensional visualization and even animation. We do not describe these visual representations in detail, rather we consider factors that relate software visualizations to particular perspectives. As shown in Fig. 2, there is a significant difference in popularity of these visual representations. Node-link layout and line representation are still the most popular metaphors despite they have notable limitations in visualizing large scale program. 3D cityscape views and even n-dimensional representations have been proposed with the development of 3D technology. However, further research is needed for addressing the weakness of this metaphor.

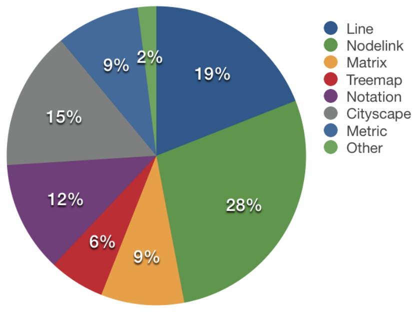

Figure 2. Number of studies per metaphor

TABLE IV. Number of papers associated to each analysis task and visual metaphor

Tasks	**Line**	**Node link**	**Matrix View**	**Treemap**	**Notation**	**Cityscape**	**Metric View**
control-flow analysis			1		5	1	
code map		5			3		
dependency relationship		15	5	2	3	7	2
software architecture		10	1	4		2	1
code metric	5	1			1	1	3
executed time	3						
statement coverage	6	1					
dynamic architecture		2					
program slice	1			1			
evolution metrics	5	1				3	5
software archives	2	1					
structural change		1	2			2	

C. Relationship between Analysis Task and Visual Metaphor

It makes no sense to translate mere source code information into a massive graph. The utility of visualization lies in an appropriate, understandable and effective abstraction of the data in order to present significant information[20]. An increasing number of visual metaphors have been proposed to address different concerns which have been reported above. In order to develop suitable SV techinques, it is necessary to understand the correlation between analysis task and visual metaphor.

Table IV has listed the number of studies which mentioned visual metaphors for respective tasks. There is a many-to-many relationship between analysis task and visual metaphor. As little attention has been paid on the visualization of control-flow analysis and code map, metaphors for these tasks are relatively few. Notation view is the most popular representation which has been employed to visualize control-flow of code. In contrast, visual representations have supported dependency relationships and architecture quite a lot. This is due to the importance of understanding structural elements for software maintanence. Common method to visualize relationship among components is graph. In particular, node-link layout has been widely used in dependency and architecture domains with 15 and 10 studies respectively. Because of increasing relations in code, matrix views which address space problem of node-link layout have been used for visualizing dependency in 5 studies. And cityscape views provide more vivid representation in comparison with previous two representations. Treemap is also applied to visualize relationship and software architecture, especially good at representing hierarchical information.

Few studies described the visualization of dynamic analysis. Among them, line representation is dominant metaphor which is applied for the visualization of dynamic aspect while node-link layout is rarely used.

With respect to the evolution, line representation and metric views are popular metaphors to depict change of metrics. While node-link and treemap can be used to visualize changes in structure.

IV. Conclusion and Future Work

This paper presents a systematic review of code visualization which is intended to provide an understanding of current metaphors used for tasks. However, there still exists limitations for our review. First of all, we focused on English articles in six famous public venues and four large digital database which excluded several valuable literature. Secondly, the variants of search terms might cause the missing of articles. Therefore, visual concerns that we consider do not exhaust all the possibilities. Instead, they are examples to illustrate certain problems and represent most popular concepts in our review.

We have specified 3 categories of visual tasks and synthesized 7 types of metaphors from SLR. However, it is argued that what is visualized is what can be visualized, not necessarily what needs to be visualized in peer academic literature[21]. Most studies try to develop new visualization techniques as oppose as to validate or add value to existing ones. This implies importance to move from the state of the art to state of practice. Simple mapping between them has provided theoretical evidence for deeper quantitative analysis. Consequently, to evaluate the utility of visualization metaphors and select appropriate one for specific task, next step is to make research on the higher level of abstraction with experimentation in cognitive and perceptive activities.

References

[1] A. Telea, L. Voinea, and H. Sassenburg, "Visual tools for software architecture understanding: A stakeholder perspective," *IEEE software*, vol. 27, no. 6, pp. 46–53, 2010.

[2] I. Spence, "Visual psychophysics of simple graphical elements." *Journal of Experimental Psychology: Human Perception and Performance*, vol. 16, no. 4, p. 683, 1990.

[3] S. Diehl, *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer, 2007.

[4] H. A. Duru, M. P. Çakır, and V. İşler, "How does software visualization contribute to software comprehension? a grounded theory approach," *International Journal of Human-Computer Interaction*, vol. 29, no. 11, pp. 743–763, 2013.

[5] V. R. Basili, J. Heidrich, M. Lindvall, J. Münch, M. Regardie, D. Rombach, C. Seaman, and A. Trendowicz, "Linking software development and business strategy through measurement," *arXiv preprint arXiv:1311.6224*, 2013.

[6] S. Keele, "Guidelines for performing systematic literature reviews in software engineering," Technical report, EBSE Technical Report EBSE-2007-01, Tech. Rep., 2007.

[7] T. Khan, H. Barthel, A. Ebert, and P. Liggesmeyer, "Visualization and evolution of software architectures," in *OASIcs-OpenAccess Series in Informatics*, vol. 27. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[8] "Visual studio 2013," http://msdn.microsoft.com/zh-cn/library/dd831853.aspx, accessed: Sept. 22, 2014.

[9] "Visualizing and understanding code," http://msdn.microsoft.com/zh-cn/library/dd409365.aspx, accessed: Sept. 22, 2014.

[10] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *ACM Sigplan Notices*, vol. 8, no. 8, pp. 12–26, 1973.

[11] T. Ball and S. G. Eick, "Software visualization in the large," *Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[12] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr, "Seesoft-a tool for visualizing line oriented software statistics," *Software Engineering, IEEE Transactions on*, vol. 18, no. 11, pp. 957–968, 1992.

[13] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," in *Proceedings of the 4th international workshop on principles of software evolution*. ACM, 2001, pp. 37–42.

[14] W. Randelshofer, "Visualization of large tree structures," 2011.

[15] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster, "Visualizing software changes," *Software Engineering, IEEE Transactions on*, vol. 28, no. 4, pp. 396–412, 2002.

[16] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*. IEEE, 2007, pp. 92–99.

[17] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz, "Software landscapes: Visualizing the structure of large software systems," in *Proceedings of the Sixth Joint Eurographics-IEEE TCVG conference on Visualization*. Eurographics Association, 2004, pp. 261–266.

[18] M. Clauß, "Generic modeling using uml extensions for variability," in *Workshop on Domain Specific Visual Languages at OOPSLA*, vol. 2001, 2001.

[19] T. Barlow and P. Neville, "A comparison of 2-d visualizations of hierarchies," in *Information Visualization, IEEE Symposium on*. IEEE Computer Society, 2001, pp. 131–131.

[20] M. Petre, E. de Quincey *et al.*, "A gentle overview of software visualisation," *PPIG News Letter*, pp. 1–10, 2006.

[21] M. Petre, "Mental imagery and software visualization in high-performance software development teams," *Journal of Visual Languages & Computing*, vol. 21, no. 3, pp. 171–183, 2010.