International Journal of Software Engineering and Its Applications
Vol.12, No.3 (2018), pp.39-50
http://dx.doi.org/10.21742/ijseia.2018.12.3.04

Evaluating the Role of Software Visualization Techniques as Assistant Tools in Software Reverse Engineering

Abdullah O. Al-Zaghameem

Department of Computer Science, Tafila Technical University
Postal: 66110, P.O. Box: 179, Tafila, Jordan
aoz@ttu.edu.jo

Abstract

Software reverse engineering is an important process for software maintenance and upgrading. The process includes analyzing existing software to identify its structural components and the relationships between them. Presenting software components, relationships, and data to the reverse engineer visually is considered as essential and vital technique. This paper explores and evaluates the role of software visualization techniques in software reverse engineering process. An evaluation model is constructed, which consists of quantitative and qualitative measurement sets. The evaluation model is applied on six software visualization tools that are used to reclaim the structural design of object oriented software. The results reveal a real need to enhance software visualization tools to play a better role in software reverse engineering.

Keywords: *Software Visualization, Reverse Engineering, UML Diagrams, Software Redesign, OOM*

1. Introduction

Software visualization (SV) is concerned mainly with the static and dynamic visualization of software components such as executable programs and source code [1, 2]. It can range from simple SV techniques that visualize system components textually to advanced SV models that exploit more advanced visualization mechanisms such as 3D Modeling [3-5]. In practice, SV techniques are used to reduce the complexity of systems for better understanding and being used more efficiently. Since the invention of Graphical User Interfaces (GUI) in the early 1980s [6], SV techniques have been widely used in several vital areas such as teaching, presentation, and designing of complex systems. In addition, SV techniques have been exploited in many software domains such as reverse engineering, reengineering, and maintenance, where a huge amount of complex data needs to be understood [7, 8].

Reverse engineering, on the other hand, is the process of analyzing existing software to identify its main components and the relationships between them in order to provide high abstraction levels for the system. It can be seen as going backwards through the cycle of software development from implementation (source code or binary code) to the analysis. Reverse engineering plays a major role in software reuse and maintenance. Practically speaking, the main goal of reverse engineering is to represent the subject system in an abstract representation in order to be more understandable by the engineer. The process of reverse engineering is a two-step process: (1) the extraction of system information through system analysis and (2) abstraction of system components by creating views and documents [7, 9].

However, reverse engineering is a very challenging process [10, 11]. From one side, it may take weeks or months to obtain a mental model of the subject system. From the other

Received (July 8, 2018), Review Result (September 17, 2018), Accepted (October 4, 2018)

ISSN: 1738-9984 IJSEIA

side, software engineer may spend a lot of time in reading and understanding software in hand [12]. For example, understanding the object-oriented source code of a system is more difficult than understanding other programming types because of the difficulties that may result from the technical aspects of object oriented model such as polymorphism and inheritance. The situation becomes worse in the case of legacy software that their source code is unavailable or missing.

Since reverse engineering is usually done under time pressure, reading and understanding the source code is not viable [11] especially for complex systems; therefore, presenting data to the reverse engineer in a suitable manner, which speeds up the reengineering process, is an important issue. For this purpose, the research community of software engineering and re-engineering has proposed a lot of solutions to solve this problem or reduce its consequences. One of the promising proposals that have been presented is the SV technique, which is considered as a comprehensive aid.

This paper figures out some key features of SV techniques and emphasizes their role in software reverse engineering. An evaluation model is constructed to evaluate this role and applied on a case study. The paper is organized as follows: Section 2 gives a background on SV. Section 3 explains the motivation and goals of this research. Section 4 discusses the methodology on which the research is conducted, and presents the evaluation model with which SV role is evaluated. In Section 5, an empirical study is conducted to apply the evaluation model on a set of SV tools and discusses the results in detail. Related works are presented in Section 6, and Section 7 concludes the work and presents some future works.

2. What is Software Visualization?

Visualization is defined by Gershon as “the process of transforming information into a visual form, enabling users to observe the information” [1]. This technology of information representation has been heavily used in mechanical engineering, chemistry, physics, and medicine [2]. Stephan Diehl [2] defines Software Visualization (SV) as “the visualization of artifacts related to software and its development process”. These artifacts may include program codes, design models, requirements documentation, *etc*.

At the structural level of software, it is important to emphasize software components and the inter-relationships that “gluing” them altogether. In this context, SV techniques are expected to play a primary role. For example, modeling languages like Unified Modeling Language (UML) offer variant sets of structure, behavior and component modeling diagrams [13]. In addition to the familiar structure visualization using UML, other SV techniques provide different methods of representation. In [2], Stephan had presented a short description of some taxonomies and surveys that classify SV techniques. This issue, however, is beyond the scope of this paper.

3. Motivation and Research Objectives

The main objective of this paper is to investigate and evaluate the role of exploiting SV techniques in software reverse engineering process. The research is motivated by inspecting the importance of SV in presenting information more precisely and more clearly than, for example, traditional textual way. In the scope of software reverse engineering, SV tools will shorten the distance between reverse engineers and the better understanding of structure and functionality of software systems. In addition, SV tools can speed up the process of software redesign, refactoring, and maintenance; especially with the employment of model-based software generators.

Few researches have discussed or considered evaluating the role of SV tools in reverse engineering in a systematic way. The research aims to add a contribution in this direction.

4. Research Methodology Plan

To emphasize the role of SV tools in reverse engineering of software, this paper studies the capabilities of several relative SV tools. To achieve this goal, the research stands mainly on conducting a kind of diagnostic comparison between these tools to evaluate their assistance in reverse engineering. Figure (1.a) illustrates the research methodology plan, which consists of two main phases. The first phase is titled *reverse engineering process*, in which SV tools are used to reclaim the design of a specific system from its existing source/binary code. In the second phase, which called *analysis and assessment process*, a detailed analysis will be made to study the resulted designs and evaluating the tested SV tools according to two sets of measurement criteria; as shown in Figure (1.b).

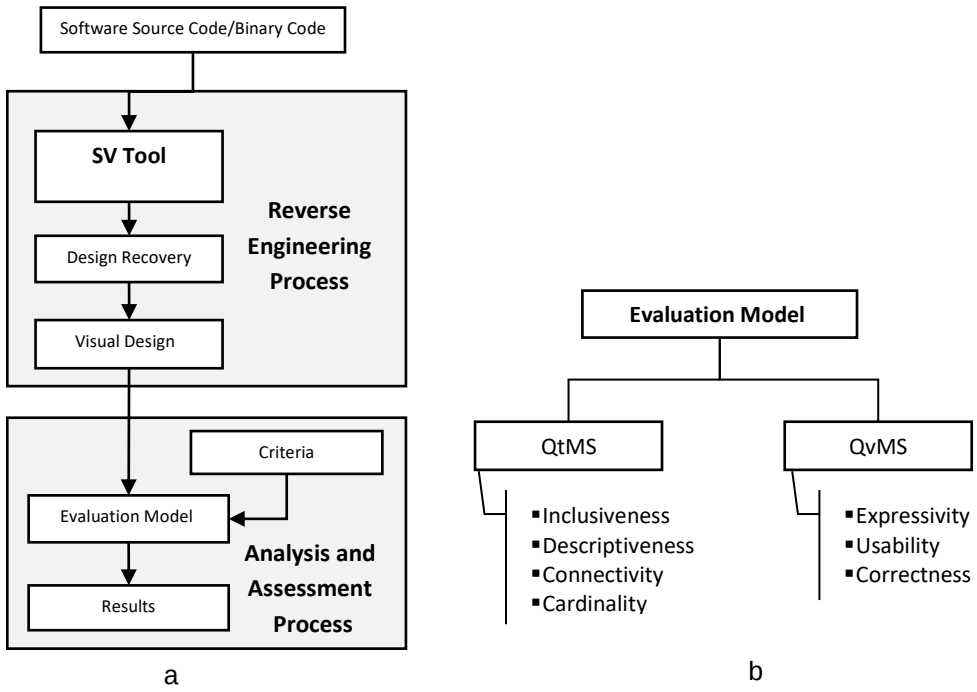

Figure 1. (a) Research Methodology Plan, and (b) Measurement Sets of the Evaluation Model

4.1. Reverse Engineering using Visualization Techniques

The reverse engineering of software using SV tools addresses mainly the structure of that software. In this research, the UML Diagrams have been selected to represent software structures visually. UML is the de facto of software modeling and brings significant benefits to software engineering and system maintenance [14]. Consequently, the selected SV tools are required to produce a UML class diagram of a specific system from the delivered source/binary code. In the context of reverse engineering, an efficient SV tool should successfully reclaim the design of system structures with an accepted level of details that can help software engineers. Moreover, the SV tool should present the system in a way that building units and the inter-connection links of its structures are discovered and represented clearly.

4.2. Evaluation Strategy

In order to inspect and evaluate the role of SV tools in software reverse engineering, an evaluation model has been constructed. It consists of two measurement sets; the *quantitative measurement set* and the *qualitative measurement set* (see Figure (1.b)). For

each of these sets, a group of criteria has been formulated as the minimal criteria set required to perform the included measurements.

For SV tools, the evaluation process concerns mainly with the tasks of reverse engineering regarding the conformance of reclaimed designs with the structure objectives and system requirements of software. The evaluation model targets those software systems that have been developed using the Object-oriented Model (OOM).

Quantitative Measurement Set (QtMS)

The following paragraphs discuss the criteria included in this set and present the objective(s) of each of them.

1. *Inclusiveness*. This criterion is primary, and determines whether the design that is regenerated by the SV tool includes *all* the structural units and modules of the system or not. It inspects the capability of SV tool to reclaim all structural units of the system and distinguishes their *types* (*e.g.*, Java programming language supports class, interface, and enumeration modules). To measure this criterion, the percentage of reclaimed units out of the total system units is recorded.

2. *Descriptiveness*. This criterion inspects if the regenerated design involves system components along with all their attributes and specifications represented in an official manner, in this research, according to the UML formal symbolization. In this context, extraction of the actual names of components and structural units is very important as it helps in detecting and describing the relationships between system units and their semantics. In addition, it can improve the understanding of the essence and functionalities implied in each component. Furthermore, it assists engineers in performing other processes in reverse engineering like system profiling.

3. *Connectivity*. Object-oriented programming model comprises different types of relationships that can connect system components altogether. Some of these relationships are structural like composition and aggregation, while others are functional like association. It is important for system reverse engineering to accurately detect all components interrelationships, recognize their types, and identify their multiplicity. This criterion determines if the SV tool infers *all* the relationships between system components?

4. *Cardinality*. For the sake of completeness, the regenerated design should include the same exact methods (functionalities) and attributes (specifications) of each structural unit in the system as it was originally developed. In this concern, SV tools and reverse engineers may manipulate this issue differently. For instance, some methods and attributes might be inherited from parent modules; therefore, these methods may appear in the design two times or more, or none at all depending on the SV tool strategy. This may preclude a better understanding of system structure and functionality, and causes structure ambiguity to programmers and reverse engineers. In this paper, two quantities will be measured to evaluate this criterion for each structural unit; the *number of methods (NOM)* and *number of attributes (NOA)* it implements or defines.

Qualitative Measurement Set (QvMS)

It is important to evaluate qualitatively the role of SV tools in software reverse engineering. Accordingly, this paper discusses and evaluates the following qualitative criteria:

1. *Expressivity* which points here to the qualitative measurement of SV tool against the clarity of system structure representation; which includes the context in which each system component is put in and the name each component is given. For each

component in the system, the *full-qualified name* should be reclaimed by the SV tool, and the context in which this component is resided (for example, namespace, package, *etc.*,) should be determined. To evaluate SV tools against this criteria, one of the following values is assigned:

a. *Excellent*: means *all* reclaimed system components are expressed accurately.

b. *Very Good*: if the majority of reclaimed components and structures have been expressed correctly.

c. *Good*: if most components and structures are put in their contexts and have accurate names.

d. *Satisfied*: if it is helpful to understand the structure of the reclaimed system design even though not all its components have been expressed precisely.

e. *Poor*: if it is hard to understand the reclaimed system structure because of *"bad"* naming or missed or incorrect contexts.

2. *Usability*. the usability is used here to refer to the capability of SV tool to help reverse engineers to reclaim the system structure in shorter time and less efforts than white-box approaches, for instance. Although measuring usability is a controversial issue because it is "user experience" dependent, the assessment of usability in this paper is conducted in terms of assessing some generic features of SV tools. Such features include: import source/binary code, save/open diagrams, customization tools, *etc*.

   Another qualitative usability measurement is the number of steps needed to perform design regeneration. In this regard, the minimum number of steps the SV tool performs the best in usability. The measurement scale used in the previous criterion will be used to evaluate usability.

3. *Correctness*. This criterion measures the accuracy degree of SV tool. In addition to reclaiming *all* system components and putting each of them in its accurate context, it is very important to connect them precisely. The precision here means extracting the actual number and type of relationships, as well as assuring that each relationship connects the accurate components. The measurement scale of this criterion is the same as the previous two.

5. Case Study: Reverse Engineering an Object-oriented System

In this section, an experiment is conducted to apply the evaluation model presented previously on a selected system. The experiment targets systems developed in the object oriented approach.

5.1. The "Toy-shop" System

The methodology and evaluation model are applied on a system called "Toy-shop"[2]. The system is programmed using Java™ and structured as shown in Figure 2. It is a simple system that organizes the process of toys shopping at an arbitrary toy store. It contains (19) different Java classes and interfaces. The system structural units are "glued" together using different relationships like inheritance, association, containment and aggregation. Among the 19 classes, there are four enumeration classes and two interfaces. One class is implemented as inner-class (inside another class) in purpose of testing reverse engineering the *containment* relationship.

[2] The source code is available for free at
https://www.dropbox.com/s/a8a47vt1zb64tzh/ToyShop.rar?dl=0

5.2. The Target SV Tools

A set of six different SV tools have been tested to reclaim the UML Class Diagram of the "Toy-shop" system. According to their developers, all tools can generate UML diagrams from Java code. The selected SV tools are "Code 2 UML" v0.6.1 [15], ObjectAid UML Explorer v1.2 [16], StarUML v4.8 [17], Class Visualizer v1.8 [18], Visual Paradigm v14.2 [19], and Papyrus Oxygen v3.0 [20]. There are, however, other SV software and tools that could be tested.

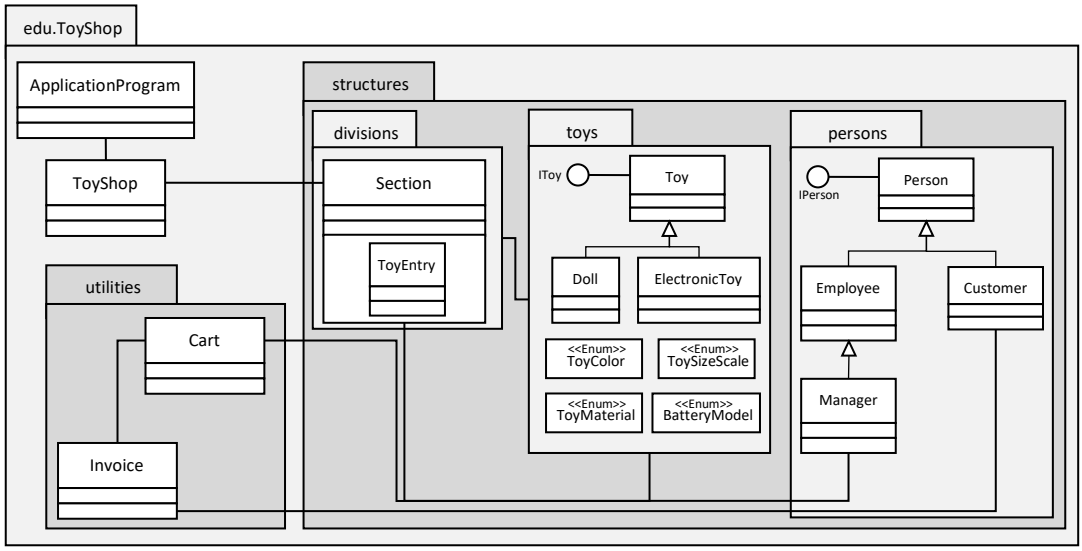

Figure 2. "Toy-Shop" System Structure Diagram

In the sake of achieving realistic results, all SV tools are executed on the same machine. A personal computer is used with Intel Core 2 Duo 2.1GHz CPU, and 4 GB DDR2 RAM. The computer operates by Microsoft Windows 7 Professional 32-bit. To verify experiment results, each SV tool has been tested twice.

5.3. Experiment, Results, and Discussion

The SV tools have been evaluated according to the evaluation model. All tools are tested according to the default settings of each one; unless it is very necessary to pre-configure some settings. First, the QtMS criteria on each recovered design have been measured for each tool and organized as shown in Table 1. As the table illustrates, all SV tools have successfully reclaimed all the (19) "Toy-Shop" classes as expected. Therefore, they all satisfy the *Inclusiveness* criterion.

Table 1. Evaluation Results of QtMS Critera

		Code 2 UML	ObjectAid UML	StarUML	Class Visualizer	Visual Paradigm	Papyrus
Inclusiveness – (19 Class)		100%	100%	100%	100%	100%	100%
Descriptiveness		83%	94%	81%	55%	93%	80%
Connectivity		62%	92%	67%	90%	91%	90%
Cardinality	NOM (91 Method)	78 (85.7%)	97 (106.6%)	70 (76.9%)	48 (52.8%)	48 (52.8%)	132 (145%)
	NOA (65 Attribute)	63 (96.9%)	65 (100%)	62 (95.4%)	98 (150.8%)	79 (121.5%)	65 (100%)

The results vary with respect to the *Descriptiveness* criterion. Although none of the SV tools reaches the100% score, some tools records good percentages. The reasons behind

that none of the SV tools scores 100% percentage include: (1) most of the tested SV tools does not obey the UML standard notations partially, or does not use them at all. For example, "Code 2 UML", StarUML, "Class Visualizer", and "Papyrus" do not enclose system classes within UML package notations. In addition, most of tested SV tools show no difference between *aggregation* and *composition* relationships; it uses the same UML link notation as in "StarUML" which uses the aggregation symbol (the famous outlined diamond) for all types of associations. Moreover, "Code 2 UML" and "ObjectAid UML" use the UML class notation for both classes and interfaces.

Another reason is that (2) some SV tools represent some system modules, surprisingly, incorrect. For example, "Class Visualizer" represented the methods declared in "IToy" and "IPerson" interfaces as attribute (field) declarations. If a decision to be made here, with these bad percentages for *descriptiveness,* software reverse engineers can go mistakenly in further steps when, for example, addition structural units to be added.

For *Connectivity* evaluation, the SV tools are tested using two main measures:

1. The relationship type. A relationship type points to the category to which the links between system structures is belong. For Java-based systems, the following relationships are considered:
    a. Association (including aggregation and composition).
    b. Inheritance.
    c. Implementation (*i.e.*, interface implementation).
    d. Confinement (*e.g.*, inner classes).
2. The number of relationships. For each tested SV tool, the count of relationships among structure units is recorded.

The results of *connectivity* evaluation are recorded and organized in Table (2). All tested tools have successfully reclaimed all *inheritance* and *implementation* relationships (as expected); because these relationships are clear, vital, and easy to recognize. In the case of *confinement* relationship, most of the tested tools mark it in some or the other way. For example, "ObjectAid UML" and "Visual Paradigm" use the famous circled plus sign (⊕) to mark confinement relationship. Others like "Papyrus" represent the inner class as part of the parent. Two of the tested tools are unable to reclaim, or point out to, this relationship.

As for the association links, the results are strangely vary from one tool to another. Among the (22) associations in the original design, "StarUML" has reclaimed only (5), recording the worst percentage among the other tools. Not in a better rank than "Papyrus"; which reclaimed only (6) associations. With these percentages, software reverse engineers can make, with almost high probability, wrong decisions in the forward steps.

Table 2. Results of Connectivity Evaluation - Relationship Detection and Identification

	Code 2 UML	**ObjectAid UML**	**StarUML**	**Class Visualizer**	**Visual Paradigm**	**Papyrus**
Association + *Aggregation* (22)	11 (50%)	22 (100%)	5 (22.7%)	16 (72.2%)	14 (63.6%)	6 (27.3%)
Inheritance – (5)	5 (100%)	5 (100%)	5 (100%)	5 (100%)	5 (100%)	5 (100%)
Implement – (2 Interfaces)	2 (100%)	2 (100%)	2 (100%)	2 (100%)	2 (100%)	2 (100%)
Confinement – (1 Inner Class)	No	Yes	No	Yes	Yes	Yes

The last evaluated criterion is the *Cardinality* of system structures. The term cardinality is employed in this research to point out the number of methods and attributes implemented in the system structural units *i.e.*, Java classes. This criterion is of a great benefit to software reverse engineers to understand the context and functionality of each unit and objects behavior. As Table 1 shows, the cardinality is measured in terms of two factors:

1. Number of Methods (NOM): the research uses the term "methods" to point all implemented class methods including: member methods, class constructors, and fields' getters and setters. Generally speaking, the NOM is the total number of all methods in all classes. As seen in Table 1, none of the tested SV tools have reclaimed exactly the (91) methods which is expected for those tools that reclaimed more methods than this number. Firstly, the Java compiler injects an empty constructor method in those Java classes that have not yet implemented the empty constructor. Some SV tools take care of this issue; therefore score results that have exceeded the actual NOM value like "ObjectAid UML". Secondly, due to inheritance relationships, child classes obtain *all* its parent's methods. Only "Papyrus" represents this "obtaining" explicitly; therefore, it scores a *wrong* high value with (132) methods. The remaining tools exhibit various behaviors. Some of them like "Code 2 UML" hide all class constructors. "Visual Paradigm" and "Class Visualizer" did not reclaim all methods; in fact, some methods have been reclaimed as attributes! Finally, "StarUML" represents interfaces as closed circles (the formal UML notation of interface) with no description about the declared methods.
2. Number of Attributes (NOA): in the original system, there were (65) attributes defined in all system classes in total. The NOA refers to this value. Most of the tested SV tools reclaimed all defined attributes. However, only "Class Visualizer" and "Visual Paradigm" reclaim more attributes than expected because both tools explicitly represent the inherited attributes in child classes. In addition, the first tool has *unexpectedly* represented some methods as attributes!

The second part of the evaluation model is the QvMS. The criteria involved in this part target the performance and productivity of software reverse engineers when using SV tools. According to the observations of three software engineers who have used the tested SV tools, the evaluation results of QvMS criteria come out as illustrated in Table 3. The results listed in the table are the average of three evaluation values for each criterion.

As the table shows, none of the tested tools was perfect in total. Some tools have graded excellent in one or two criteria but scored less in other criteria. All in all, all tested tools exhibits good usability, but varies at the correctness criterion; again because not all tested SV tools obey the formal notations of UML. As could be noticed from the table, "Class Visualizer" and "StarUML" are "Poor" with respect to the *Correctness* criterion because they incorrectly reclaimed some member methods of classes as attributes. Moreover, many associations among classes are missed or represented incorrectly in the reclaimed designs.

Table 3. Results of Qualitative Evaluation

	Code 2 UML	**ObjectAid UML**	**StarUML**	**Class Visualizer**	**Visual Paradigm**	**Papyrus**
Expressivity	Good	Very Good	Good	Satisfied	Excellent	Good
Usability	Good	Excellent	Good	Good	Very Good	Very Good
Correctness	Satisfied	Excellent	Poor	Poor	Excellent	Very Good

To sum up, and from software reverse engineering perspective, using SV tools can save a huge amount of effort and time. Nevertheless, as concluded from the previous results, they put extra efforts on reverse engineers to peer inspect the reclaimed designs, and may compel the use of more than one tool to get better results. This does not disaffirm that SV tools play an important role in reverse engineering process.

6. Related Works

6.1. SV Role in Reverse Engineering and Software Comprehension

It is important to shed a light first on the use of SV tools in reverse engineering and software comprehension in research. The following paragraphs present some of research papers which are relevant to the main goal of this paper.

MJ Pacione *et. al.* [21] argued that the current SV techniques address only specific aspects of reverse engineering context, and fail to solve the difficulties that inherent from the object oriented languages nature. The authors conducted an evaluation experiments to evaluate the dynamic performance of visualization tools. Several tools were evaluated by assessing their performance in a number of visualization tasks. The evaluation model in this research focuses on the static structure of software and measures the qualitative features of SV tools to judge their role in reverse engineering in general. The results of this research and those concluded in [21] have agreed that more attention needs to be paid to SV tools development in order to improve their role in software reverse engineering.

Richard Wettel and Michele Lanza [22] proposed a 3D software visualization system based on the city metaphor. The city metaphor is useful in the field of reverse engineering and program comprehension. The authors argued that a good visualization technique can help in reverse engineering tasks by supporting habitability. Habitability is the source code characteristic that makes it easy to understand by individuals rather than developers. The proposed technique supports the habitability and locality concepts using city metaphors. The technique views the subject system as a city with classes as buildings and packages as the districts of that city. The technique is applied on two large systems called ArgoUML and Azures. ArgoUml is a java project used to generate and draw UML diagrams, and Azures is a peer-to-peer java application. The experiment approved scalability, interactivity and completeness of the proposed technique. Another research by Fittkau *et. al.* [23] proposed a technique called ExplorViz in which authors have claimed that it enhances application visualization for better software comprehension.

Mendelzon and Sametinger [24] used a general visualization tool called Hy+ to visualize information about object-oriented software systems. Hy+ is a tool used to visualize objects and relationships between them. The tool provides a user interface and supports a query language called Graphlog. The authors argued that using general purpose SV system is more flexible than using reverse engineering tools.

Rainer Koschke [25] conducted a survey on 82 researchers in SV in software maintenance, reverse engineering and reengineering. The survey answered several questions include: To what degree researchers are involved in SV? What to visualize and how? Is animation frequently used in SV? Do researchers believe that visualization is useful at all? Which automatic graph layouts are used? Do the layout algorithms have deficiencies? And finally, where should future researches directed? The survey showed that among visualization techniques, graphs are used in 52% of the researches, 18% have used UML Diagrams, and 18% used textual representation. The animation techniques are rarely used.

Michael J. Pacione [26] provided an approach to improve the effectiveness of visualization techniques by combining the structural and behavior perspectives of formal model of abstractions, the facets that represent the important aspects of the system, and the dynamic and static information that are extracted from the subject system. The paper provided practical questions that considered as a basis for visualization tool evaluation. Our research considers the work by Michael, according to the evaluation model results, as a conclusion. That is, SV tools need to provided with techniques and information sources so that visual representations become more helpful to reverse engineers. For example, this research suggests the use of annotation technique to augment visualizations with functionality modeling or execution flow controls.

6.2. Evaluating the SV Tools in Reverse Engineering and Software Comprehension

In research literature, few researches have directly presented taxonomy studies or evaluation comparisons in the role of SV tools in reverse engineering. However, the authors of [27] presented a survey on state-of-the-art information visualization techniques (InfoVis). Taxonomy has been built based on four main categories on a detailed review of the literature. These categories are empirical methodologies, interactions, frameworks, and applications. The taxonomy has been augmented with the advantages and limitations of the studied methods under each major category. A more dissemination search has been made by Liam and Laramee [28], in which a survey of surveys (SoS) has been conducted and presented as a road map that guides researchers in the field of information visualization in general.

Anna-Liisa *et. al.* [29] conducted a study that presents a systematic literature review spanning six years of software visualization literature (starting from the year 2010). The result of their study shows that the most studied topics in the past six years are related to software structure, behavior and evolution. At the same time, software process and usage are addressed only in few studies.

Researchers in [30] have presented a systematic review to study the reverse engineering techniques that focus on recovering program interactions and represent them as sequence diagrams. Several approaches have been investigated and their features, limitations, and operation have been demonstrated. Their research comes out with three very important conclusions; one of them is that expressive and extended notations need to be attached to UML sequence diagrams to allow adding more details.

7. Conclusion and Future Works

This research aims at investigating and evaluating the role of Software Visualization (SV) tools in the software reverse engineering process. An evaluation model has been constructed. The evaluation model consists of two group of measurement sets; quantitative and qualitative sets. An experiment has been conducted to evaluate a set of six different SV tools on reclaiming the UML Class Diagram of a specific object oriented application programmed with (Java).

The evaluation results are so various among the tested SV tools. From the quantitative measurements, we can conclude that SV tools play almost a good assistant role in software reverse engineering. Nevertheless, there is a real need to augment these tools with techniques and capabilities that allow for more details to appear on system designs. For instance, the SV tool should distinguish between inherited and original class methods. A capability of drawing execution paths (e.g. method calls) can help in understanding the functionality and control flow between system components.

As for qualitative measurements, most SV tools are user-friendly and exhibit good usability. More attention needs to be paid in the conformance of these tools with the standard and formal modeling notations. The software reverse engineers cannot depend on one SV tool to comprehend the structure of system on hand, which results in exerting more efforts.

For future work, more SV tools need to be evaluated on extended sets of measurements also the case study used in this research is of small-scale object oriented software. The evaluation should target more than one application with various complexities.

References

[1] J.-E. Dubois and N. Gershon, “The Information Revolution: Impact on Science and Technology”, Springer Science & Business Media, **(2013)** March 12.

[2] S. Diehl, “Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software”, 1st Edition, Springer-Verlag Berlin Heidelberg. ISBN: 978-3-642-07985-6, **(2007)**.

[3] B. Schmid, J. Schindelin, A. Cardona, M. Longair and M. Heisenberg, "A high-level 3D visualization API for Java and Image", BMC bioinformatics, vol. 11, no. 1, **(2010)**, pp. 274.
[4] R. Wettel and M. Lanza, "Visual Exploration of Large-Scale System Evolution", 15th Working Conference on Reverse Engineering, Antwerp, doi: 10.1109/WCRE.2008.55, **(2008)**, pp. 219-228.
[5] S. S. P. Shum, K.-m. Yu and K.-m. Au, "3D Fillet Solid Model Reverse Engineering from 2D Orthographic Projections", International Conference on Manufacturing Automation, Hong Kong, doi: 10.1109/ICMA.2010.15, **(2010)**, pp. 71-78.
[6] E. S. Raymond, "The Art of Unix Programming", Addison-Wesley, ISBN 0-13-142901-9, **(2003)** October.
[7] G. Canfora Harman and M. Di Penta, "New frontiers of reverse engineering", Future of Software Engineering. IEEE Computer Society, **(2007)**.
[8] B. A. Price, R. M. Baecker and I. S. Small, "A principled taxonomy of software visualization", Journal of Visual Languages & Computing, vol. 4, no. 3, **(1993)**, pp. 211-266.
[9] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley and K. Wong, "Reverse engineering: A roadmap", Proceedings of the Conference on the Future of Software Engineering. ACM, **(2000)**, pp. 47-60.
[10] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy", IEEE software, vol. 7, no. 1, **(1990)**, pp. 13-17.
[11] G. Canfora, M. Di Penta and L. Cerulo, "Achievements and challenges in software reverse engineering", Communications of the ACM, vol. 54, no. 4, **(2011)**, pp. 142-151.
[12] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen and R. Koschke, "A systematic survey of program comprehension through dynamic analysis", IEEE Transactions on Software Engineering, vol. 35, no. 5, **(2009)**. pp. 684-702.
[13] G. Booch, J. Rumbaugh and I. Jacobson, "Unified Modeling Language User Guide", 2nd Ed., Addison-Wesley Professional, ISBN-10: 0-321-26797-4, **(2005)** May.
[14] W. J. Dzidek, E. Arisholm and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of UML in software maintenance", IEEE Transactions on software engineering, vol. 34, no. 3, **(2008)**, pp. 407-432.
[15] Code 2 UML, Official Website: https://sourceforge.net/projects/code2uml/, last visited **(2018)** September.
[16] The ObjectAid UML Explorer for Eclipse, Official Website: http://www.objectaid.com/, last visited **(2018)** September.
[17] StarUML 2, Official Website: http://staruml.io/, last visited **(2018)** May.
[18] Class Visualizer, Official Website: http://www.class-visualizer.net/, last visited **(2018)** February.
[19] Visual Paradigm, Official Website: https://www.visual-paradigm.com/features/uml-and-sysml-tools/, last visited **(2018)** February.
[20] Papyrus Modeling Environment – Eclipse, Official Website: https://www.eclipse.org/papyrus/, last visited **(2018)** February.
[21] M. J. Pacione, M. Roper and M. Wood, "A comparative evaluation of dynamic visualisation tools", 10th Working Conference on Reverse Engineering, **(2003)**.
[22] R. Wettel and M. Lanza, "Visualizing software systems as cities", 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, IEEE, **(2007)**.
[23] F. Fittkau, A. Krause and W. Hasselbring, "Software landscape and application visualization for system comprehension with ExplorViz", Information and Software Technology, vol. 87, **(2017)**, pp. 259-277.
[24] A. O. Mendelzon and J. Sametinger, "Reverse engineering by visualizing and querying", Software-Concepts and Tools, vol. 16, no. 4, **(1995)**, pp. 170-182.
[25] R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey", Journal of Software Maintenance and Evolution: Research and Practice, vol. 15, no. 2, **(2003)**, pp. 87-109.
[26] M. J. Pacione, "Software visualization for object-oriented program comprehension", ICSE 2004. Proceedings of the 26th International Conference on Software Engineering, IEEE, **(2004)**.
[27] S. Liu, W. Cui, Y. Wu and M. Liu, "A survey on information visualization: recent advances and challenges", The Visual Computer, vol. 30, no. 12, **(2014)**, pp. 1373-1393.
[28] L. McNabb and R. S. Laramee, "Survey of Surveys (SoS): Mapping The Landscape of Survey Papers in Information Visualization", In Computer Graphics Forum, vol. 36, no. 3, **(2017)**, pp. 589-617.
[29] A.-L. Mattila, P. Ihantola, T. Kilamo, A. Luoto, M. Nurminen and H. Väätäjä, "Software visualization today: systematic literature review", In Proceedings of the 20th International Academic Mindtrek Conference (AcademicMindtrek'16), ACM, New York, NY, USA, (2016), DOI: https://doi.org/10.1145/2994310.2994327, **(2016)**, pp. 262-271.
[30] T. Ahmed Ghaleb, M. A. Alturki and K. Aljasser, "Program comprehension through reverse-engineered sequence diagrams: A systematic review", Journal of Software: Evolution and Process, e1965, **(2018)**.

Author

Abdullah O. Al-Zaghameem, an assistant professor at Tafila Technical University. He has been a lecturer since 2005. He was awarded his Ph.D. from the Technical University of Berlin (Germany) in September 2012. His main research interests include software engineering, distributed programming models, distributed aspect-oriented programming, and mobile computing.

