Semantic Metadata Information (SMI) Visualisation Technique Using the Integration of Ontology and UML Graph-Based Approach to Support Program Comprehension

Rozita Kadar[1*], Jamal Othman[2], Naemah Abdul Wahab[3], Saiful Nizam Warris[4]
[1,2,3,4] *Department of Computer and Mathematical Sciences, UniversitiTeknologi MARA CawanganPulau Pinang, Malaysia*

*Corresponding author:*rozita231@ppinang.uitm.edu.my*
Received Date: 10 July 2019
Accepted Date: 16 October 2019

ABSTRACT

Representing any ideas with pictures rather than words is intuitively more appealing because a visual presentation can be more readily understood than that of textual-based. Program visualisation is one of the techniques that can be used in teaching to help users in understanding how programs work. Program visualisation technique is a mental image or a visual representation of an object, scene, person or abstraction that is similar to visual perception. This technique is significant to users because the criteria of source code cannot be physically viewed. It is applicable in the process of writing programs as it helps users to understand their codes better. The purpose of program visualisation is to translate a program into a graphical view to show either the program code, data or control flow. Visualisation technique uses the capability of human visual system to enhance program comprehensibility. Thus, this study uses program visualisation technique to represent program domain in a graphical view to help novices in improving their comprehension. This research aims to support beginners or novice programmers who have been exposed to programming languages by providing effective visualisation technique.

Keywords: *Program Visualisation, Program Comprehension, Ontology, UML Diagrams, Source Code*

INTRODUCTION

Program comprehension is necessary in performing maintenance tasks and mainly takes place before changing any process. Software maintainers must be familiar and comprehend the parts of source code in the program to be maintained (Alhindawi, Alsakran, Rodan, & Faris, 2014). However, most software maintainers face a problem in comprehending a software system while implementing maintenance tasks (Alhindawi et al., 2014).

Source code is an essential artefact for software maintainers to become familiar with a software system (Carvalho, 2013; Corley, Kammer, & Kraft, 2012; Cornelissen, Zaidman, Society, & van Deursen, 2011; Sharafi, 2011; Tiarks, Röhm, & Roehm, 2013; Yazdanshenas & Moonen, 2012). Nowadays, the expansion in size and difficulty of software system led to difficulties in maintaining the system. Software maintainers have to maintain a huge size of source code that needs to be comprehend (Haiduc, Aponte, & Marcus, 2010; Ishio, Etsuda, & Inoue, 2012) and identify a corresponding piece of code that needs to be maintained (Carvalho, 2013; Roehm, Tiarks, Koschke, & Maalej, 2012; Ying & Robillard, 2011). Among their tasks is to understand the source code before implementing maintenance tasks(Carvalho, 2013; Roehm et al., 2012; Ying & Robillard, 2011). Over the past few years, researchers have proposed various forms of graphical representation in representing text-based software system. This visualisation approach has been seen to facilitate the understanding of program domain. Program visualisation technique is

considered very important for the users since it provides mental models of an information. According to Khan & Khan (2011), graphical representation is the best way to convey complicated ideas clearly, precisely and efficiently (Khan & Khan, 2011). It makes huge and complex information intelligible.

The basic purpose of program visualisation technique is to create interactive visual representations of the information that exploit human's perceptual and cognitive capabilities of problem solving. Visualisation is used to present huge amount of information coherently and compactly from different viewpoints besides providing several levels of detail. The goal of visualisation is to help users in understanding and interpreting huge and complex sets of information (Khan & Khan, 2011). Therefore, this study explores the importance of program visualisation technique in improving source code comprehension. It proposes the use of ontological concepts as knowledge representation integrated with UML Class Model to provide more valuable information to users.

The paper is organised as follows. The next section reviews previous studies by comparing the techniques in this area. The following section discusses the proposed work. The conclusion of this study is presented in the final section.

INTEGRATION OF ONTOLOGY AND UML CLASS-BASED MODELLING

In populating an ontology, a source code should be analysed through a process of information extraction. Information extraction is a process of obtaining information in a source code and displaying it in a different view. After the information from the source code is retrieved, it should be stored in a standard form of information. It will then go through a process that uses an ontological approach. In general, ontology development is divided into two main phases; specification and conceptualisation. The goal of specification phase is to acquire informal knowledge on the domain while the goal of conceptualisation phase is to organise and structure the obtained knowledge as well as the use of UML class model. There are almost as many graphical representations presenting source code. The most common high-level representation is graph; for instance, dependency graph, data and control flow graph. Although the dependency graph has been widely used in representing a relationship, it is often more complex than the original source code artefacts. It makes them less suitable to directly support the comprehension or visualisation (Krinke, 2004; Yazdanshenas & Moonen, 2011). Thus, in this subsection discusses previous works that used the ontology representation in software comprehension, UML class model and the integration of both ontology and UML class model.

A Unified Modelling Language (UML)-based diagram is another representation that is frequently used (Bhagat, Kapadni, Kapadnis, Patil, & Baheti, 2012; Ibrahim & Ahmad, 2010; Jali, Greer, & Hanna, 2014; Kothari, 2012; Shinde, 2012; Thakur & Gupta, 2014). UML has become a de facto standard language for expressing artefacts used and produced within a software development process. UML consists of various graphical notations, describing the syntax of the modelling language and meta models, describing the semantics of UML. UML diagrams enable developers to specify, visualise, construct and document the artefacts of a software system.

UML class diagram is among the frequently used software representations to describe the structure of software systems that aid reverse engineering project. UML class diagrams allow for modelling, in a declarative way, the static structure of an application domain in terms of concepts and relations between them. In a class diagram, classes are represented by boxes with three parts;the first part contains the name of the class, which has to be unique in the whole diagram. The second part contains the attributes of the class, each specified by their name, type and visibility, whereas the operations of the class are denoted by name, argument list, return type and visibility.

Ontology representation is used to define sets of concepts describing the domain knowledge and allowing specification of classes by rich and precise logical definitions. The basic idea of using ontology for software system is to provide an artefact consisting both code knowledge and domain knowledge with which software maintainers can understand the features of source code. Ontology includes the concepts, relationships and instances to describe the specific domain of concern. Concepts are referred to as a category that is also known as a class. A series of concepts represents the topics or characters in a domain ontology; relations show the connection between concepts and used to describe the association between concepts when considering a specific concept, which is also called an attribute; while instances describe a series of concepts and relationships with specific knowledge. Instances in ontology are the values of attribute in the class that describe necessary properties. Instances also inherit all attributes or relationships of their class. At present, ontology is used in many fields to represent knowledge and provide a formal way to define the concept. Ontology has been shown to support program understanding (Wilson, 2010).

The proposed approach was used to extract an ontological point of view for software system by integration utilising the ontology and UML class-based modelling. Due to similar features shared by both UML class diagram and ontology, class diagram is used to aid the population of code ontology. The similarity of UML class model and ontology model are listed in Table 1 below:

Table 1: Concepts Similarity Of UML Class And Ontology Model

Num.	Similarity Description
1	UML class denotes a set of objects with common features, while concept in ontology also does the same thing.
2	UML class has hierarchical structure, while hierarchical structure is basic structure for taxonomy, which is one of the features that ontology has.
3	UML class has properties, while ontology has two types of properties: object property and data type property
4	UML class has relations such as associations and dependencies, while these relations represented as roles or properties in ontology.
5	Class diagram includes class name, attributes and operation, while in ontology includes concepts, relationships and instances
6	Class itself will be transformed into concept in the ontology
7	The attributes of the class will be transformed into properties of that concept in ontology
8	For the generalization classes, the relationships SubClassOf will be preserved by the subclass concepts.
9	For the inheritance classes, the relationships SuperClassOf will be preserved by the superclass concepts.
10	Association transformed into ConnnectTo property, and it is a symmetric property.
11	Dependency transformed into DependOn property and its inverse property Depend.
12	Aggregation transformed into HasA property.

13	Composition transformed into PartOf property.

Hence, a set of transformation rules was proposed to populate the ontology from a UML class diagram. On the other hand, it is believed that class diagram also has semantics representation, which is somehow implicitly preserved. Thus, ontology could be used to recover the semantics for class diagram.

SEMANTIC METADATA INFORMATION (SMI) VISUALISATION DEVELOPMENT

The purpose of creating a graphical representation or interface is to help users in utilising a system without spending much time to learn it. The technique proposed for this study is Semantic Metadata Information (SMI). In this study, visualisation technique was applied in SMI tools where the visual diagram was used to represent the source code to show the concepts and relations among concepts. Besides, the application of simple notation was used to avoid various forms of diagram that can be confusing. Furthermore, the combinations of colour were used to discriminate the structure in a program.

In addition, the zooming method is applied in this technique. This method helps to increase users' understanding on the viewable source code. Using simple diagrams provides a clearer view of a program structure.

As shows in Table 2 is the symbols that represent the concepts in ontology extracted from source code namely ***Concept Notations (CN)***. The notations are filled with colours to help users in differentiating among concepts. This study categorised the concepts into access control, class level, method level, variable, data type and variables.

The relationships among concepts were constructed called ***Relationship between Concepts Notations (RCN)***. All the relations were based on UML Class Model. There were eight types of relations proposed in this work. The details of relations are shown in Table 3.

A diagram layout for SMI is in a form of graph, $G = (N, E)$, where N is a set of objects called nodes or vertices, and E represents a set of edges. Graphs are generally represented by means of diagrams in which vertices are represented by small circles or dots as well as edges by line segments. If the graph is directed, then the edges are represented by arrows. The edges of a graph can have weights that could represent the distance between the nodes, time taken to traverse that link, probability that the link does not fail or any other measures relevant to the problem at hand. If the edges have weights, the graph is said to be a weighted graph and is represented by a triplet $G=(N,E,W)$ where W is the weight on the edges.

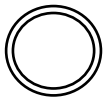
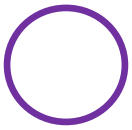
In this study, weight represented the name of relationship between two nodes, $G= (N,E,R)$.

Formal Definition: *A graph G can be defined as a pair (N,E), where N is a set of nodes, and E is a set of edges between the vertices.*

SMI diagram was designed to reduce the number of intersection (Figure 1). The diagram proposed the combination of edges that run the same node. Diagram with many edges usually distracts viewers and can lead to irregularities in the diagram. Therefore, the layout was restructured to avoid intersection.

The corner of edges was rounded to allow viewers to easily follow the path as it suits the natural movement of the viewers' head and eyes, respectively. The connection of nodes was not very concise in the case that the corners are not rounded. This design reduced the amount of space needed by the edges on the diagram, thus improving the lucidity of the diagram and aided the viewers in following the edges.

Table 2: Concept Notations (CN)

Concept: Access Control			
- Private	* Protect	+ Public	~ Default
Concept: Class Type			
 Local Class	 Abstract Class		
Concept: Field			
 Instance Field	 Static Field	 Local Field	 Reference Field
Concept: Local Variable			
 Parameter			
Concept: Method Type			
 Abstract Method	 Constructor	Instance Method	 Static Method
Concept: Data type			
 Complex	 String	 Number	 Boolean	 Array

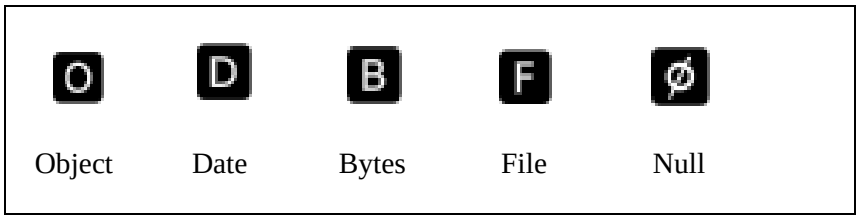
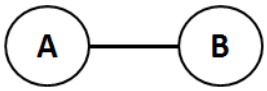
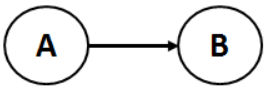
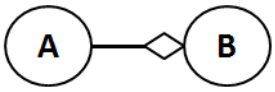
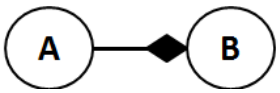
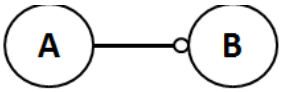
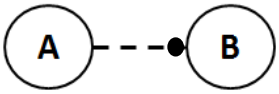
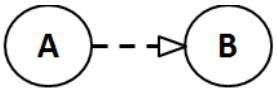
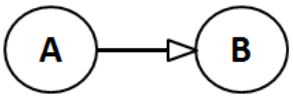


Table 3: Relationship between Concepts Notations (RCN)

Graphical Notation		Semantic Explanations
	Solid line	Class A *is associated with* class B
	Arrow	*Navigate from* class A *to* class B or Class A association class B
	Solid line with hollow diamond	Class A *composed without belonging to* class B or Class B aggregation class A
	Solid line with filled black diamond	Class A *compose and are contained by* class B or Class A has Class B or Class B composition class A
	Solid line with small hollow circle	Class A *has interface* B
	Solid line with small filled black circle	Class A *is dependent upon* class B
	Dotted line with hollow triangle	Class A *is a realization of* class B or class A uses class B
	Solid line with hollow triangle	Class A *inherits from* class B or Class A is a class B

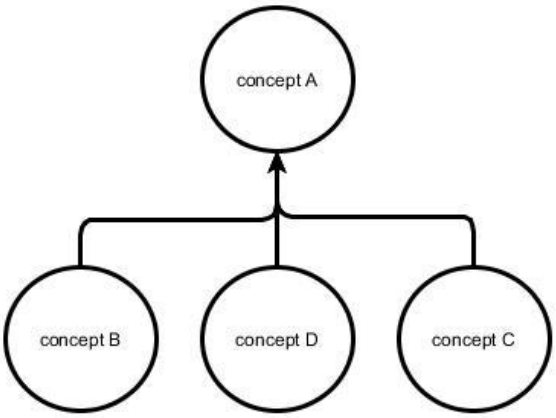


Figure 1: Reduction of edges in SMI diagram

Figure 2 depicts an example where the position labels of the node and edges were placed horizontally, which means that the text was placed horizontally and the font was the same throughout the diagram so that it can be easily read.

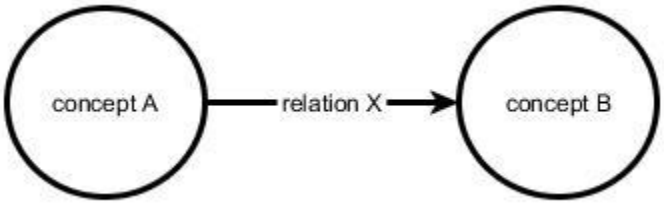
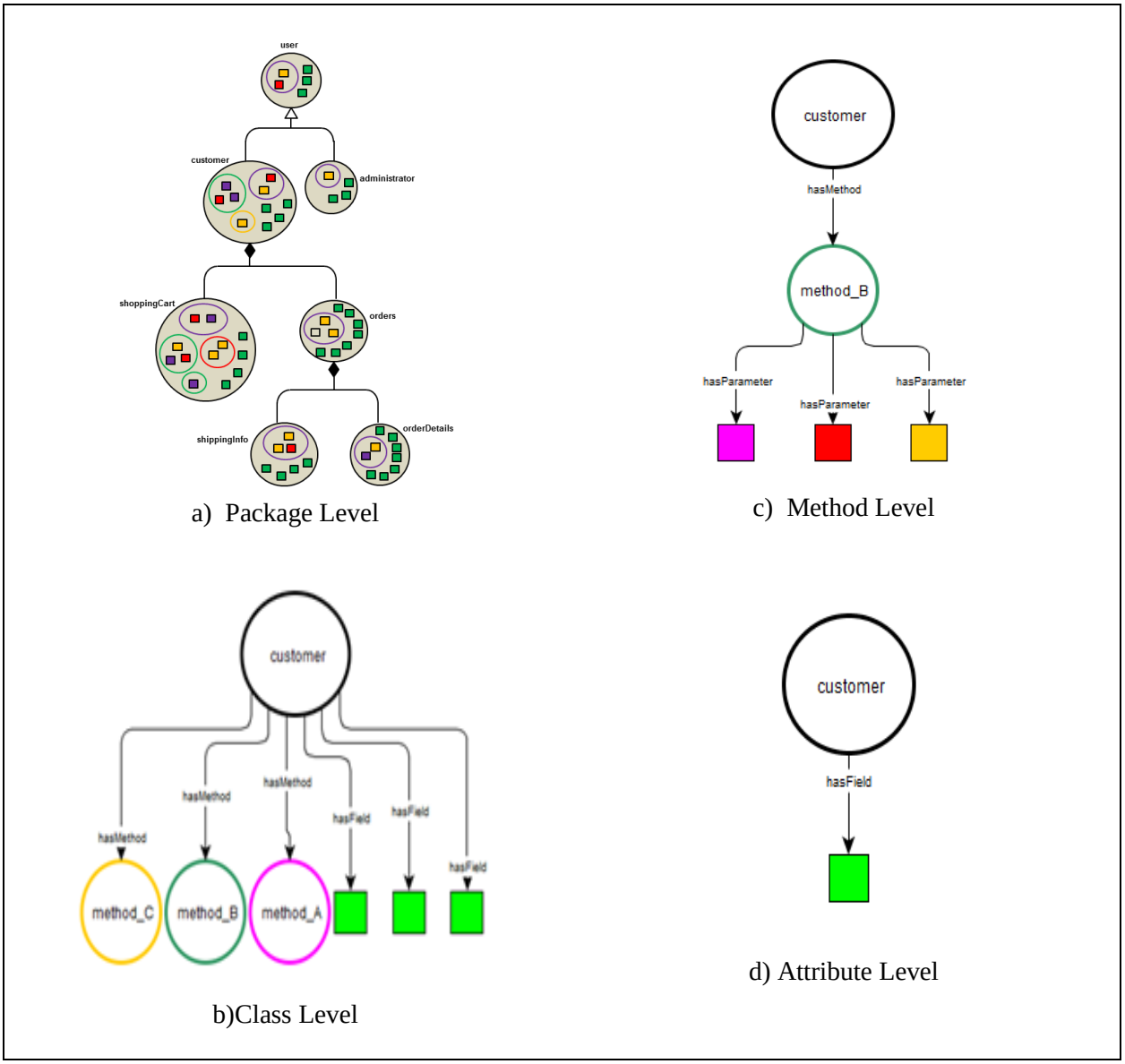


Figure 2: Draw labels horizontally

The various source code level granularities are presented in detailed from Figure 3(a) until Figure 3(d). Figure 3(a) illustrates the structure of package level granularity where it consists of 7 classes namely**user, customer, administrator, shopping cart, orders, shoppingInfo and orderDetails**. Each class consists of methods, variables and fields. Moreover, the figure shows the relationship among classes that involve the **aggregation** and **composition** relationships. Figure 3(b) illustrates the details of **customer**class that comprised methods and its fields, while Figure 3(c) shows the method level granularity that consists its parameter. The last figure (in Figure 3(d)) presents the attribute level where it shows the **customer** and its field.

As an example, Figure 4 displays the customer class that contains all methods, parameters and fields for each method. This view used the zooming method to show the details of customer class. The diagram layout was in graph-based to show the relationships of the entities and attributes.



a) Package Level

c) Method Level

b)Class Level

d) Attribute Level

Figure 3: The SMI diagrams in source code level granularity

The user interface of SMI prototype is illustrated in Figure 5. It consists of four different windows with the main windows consisting menu bar and standard toolbars. The second windows provided the space where users can write a program. The next windows place the visualisation diagrams that represent a

program source code and will pop-up after users generated the SMI tool. The tool also provided a zooming method to view in detailed the structure of source code to any level granularity.

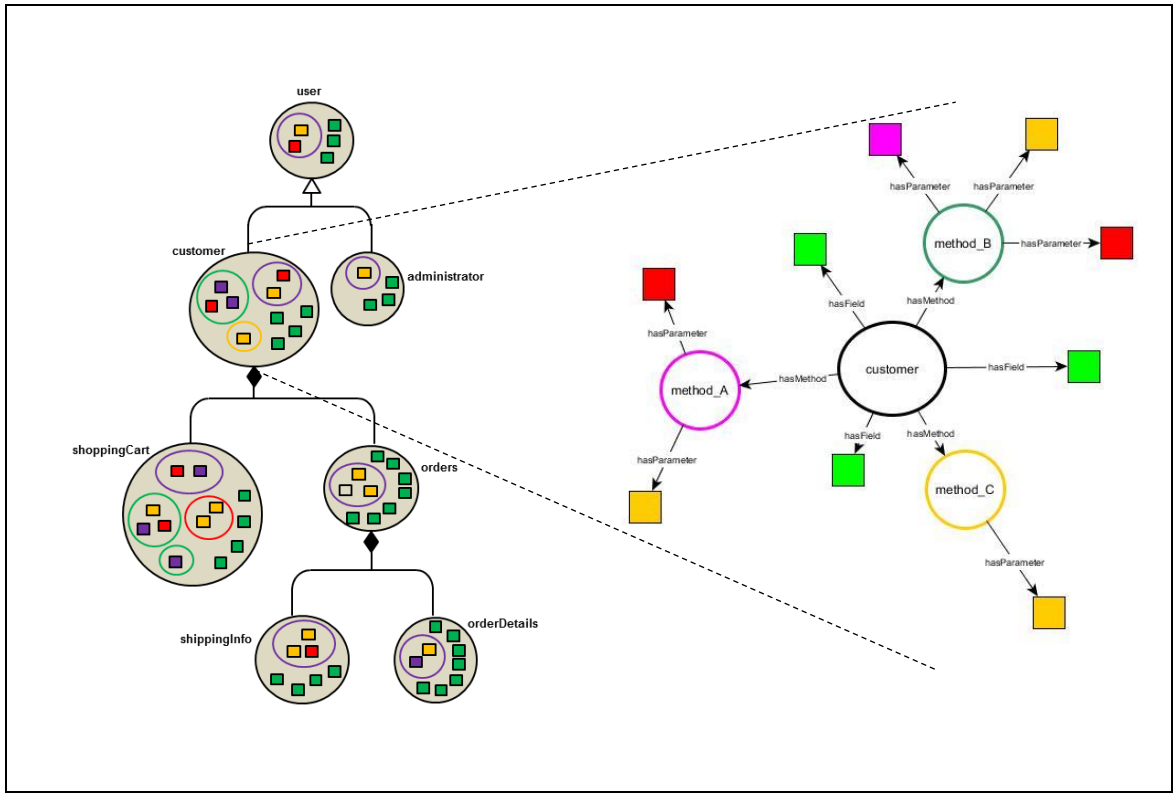


Figure 4: An example of SMI diagram of Online Shopping System using zooming method

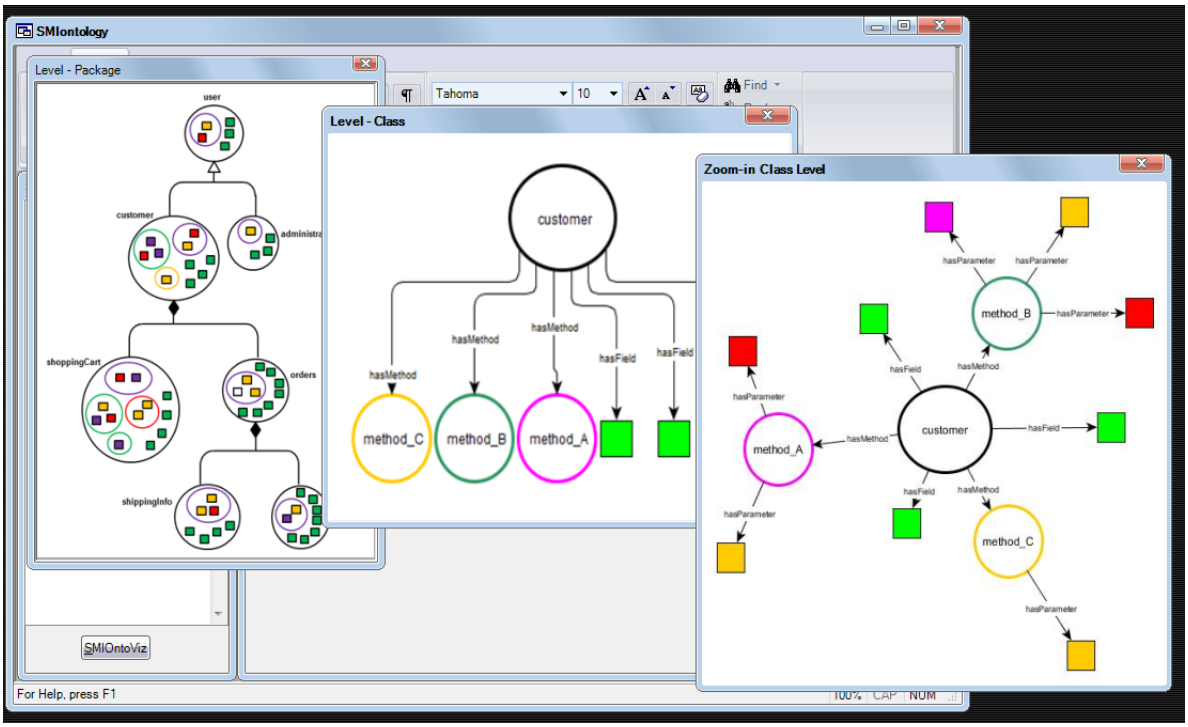


Figure 5: An example of SMI tool generated the graphical view of source code

CONCLUSION AND FUTURE RESEARCH

A suitable visual representation must be considered carefully to ensure the effectiveness of visualisation viewed. Program visualisation is important in comprehending a program as program is text-based and is not physically shown. Moreover, program visualisation helps the users to understand the behaviours of a program due to difficulties that may appear when theytry to understand the program.

This study has proposed a suitable program comprehension technique that aims to facilitate developers during maintenance. This proposed work involved extracting information from source code and constructing it into graphical view. It constituted the ontology where users can retrieve the information about a source code. Information in a form of semantic metadata as the output has been provided to users. As future work, a case study will be conducted to evaluate the effectiveness of the proposed work to support program comprehension.

REFERENCES

Alhindawi, N., Alsakran, J., Rodan, A., & Faris, H. (2014). A Survey of Concepts Location Enhancement for Program Comprehension and Maintenance. *Journal of Software Engineering and Applications*, *07*(05), 413–421. https://doi.org/10.4236/jsea.2014.75038

Bhagat, S. B., Kapadni, P. R., Kapadnis, N., Patil, D. S., & Baheti, M. J. (2012). *Class Diagram Extraction Using NLP*. 1–4.

Carvalho, N. R. (2013). An ontology toolkit for problem domain concept location in program comprehension. *Proceedings of the 2013 International Conference on Software Engineering*, 1415–

1418. https://doi.org/10.1109/ICSE.2013.6606731

Corley, C. S., Kammer, E. a., & Kraft, N. a. (2012). Modeling the ownership of source code topics. *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 173–182. https://doi.org/10.1109/ICPC.2012.6240485

Cornelissen, B., Zaidman, A., Society, I. C., & van Deursen, A. (2011). A controlled experiment for program comprehension through trace visualization. *Software Engineering, IEEE Transactions On*, *37*(3), 341–355.

Haiduc, S., Aponte, J., & Marcus, A. (2010). Supporting program comprehension with source code summarization. *Software Engineering, 2010 ACM/IEEE 32nd International Conference On*, *2*, 223–226. IEEE.

Ibrahim, M., & Ahmad, R. (2010). Class Diagram Extraction from Textual Requirements Using Natural Language Processing (NLP) Techniques. *2010 Second International Conference on Computer Research and Development*, 200–204. https://doi.org/10.1109/ICCRD.2010.71

Ishio, T., Etsuda, S., & Inoue, K. (2012). A lightweight visualization of interprocedural data-flow paths for source code reading. *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 37–46. https://doi.org/10.1109/ICPC.2012.6240506

Jali, N., Greer, D., & Hanna, P. (2014). *Class Responsibility Assignment (CRA) for Use Case Specification to Sequence Diagrams (UC2SD)*. 2–7.

Khan, M., & Khan, S. S. (2011). Data and information visualization methods, and interactive mechanisms: A survey. *International Journal of Computer Applications*, *34*(1), 1–14.

Kothari, P. R. (2012). *Processing Natural Language Requirement to Extract Basic Elements of a Class*. *3*(7), 39–42.

Krinke, J. (2004). Visualization of program dependence and slices. *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 168–177. https://doi.org/10.1109/ICSM.2004.1357801

Roehm, T., Tiarks, R., Koschke, R., & Maalej, W. (2012). How do professional developers comprehend software? *Proceedings of the 34th International Conference on Software Engineering*, 255–265. https://doi.org/10.1109/ICSE.2012.6227188

Sharafi, Z. (2011). A Systematic Analysis of Software Architecture Visualization Techniques. *2011 IEEE 19th International Conference on Program Comprehension*, 254–257. https://doi.org/10.1109/ICPC.2011.40

Shinde, S. K. (2012). *NLP based Object Oriented Analysis and Design from Requirement Specification*. *47*(21), 30–34.

Thakur, J. S., & Gupta, A. (2014). Automatic generation of sequence diagram from use case specification. *Proceedings of the 7th India Software Engineering Conference on - ISEC '14*, 1–6. https://doi.org/10.1145/2590748.2590768

Tiarks, R., Röhm, T., & Roehm, T. (2013). Challenges in Program Comprehension. *Softwaretechnik-Trends*, *32*(2), 19–20. https://doi.org/10.1007/BF03323460

Wilson, L. A. (2010). Using ontology fragments in concept location. *Software Maintenance (ICSM), 2010 IEEE International Conference On*, 1–2. https://doi.org/10.1109/ICSM.2010.5609555

Yazdanshenas, A. R., & Moonen, L. (2011). Crossing the boundaries while analyzing heterogeneous component-based software systems. *Software Maintenance (ICSM), 2011 27th IEEE International Conference On*, 193–202. IEEE.

Yazdanshenas, A. R., & Moonen, L. (2012). Tracking and visualizing information flow in component-based systems. *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 143–152. https://doi.org/10.1109/ICPC.2012.6240482

Ying, A. T. T., & Robillard, M. P. (2011). The Influence of the Task on Programmer Behaviour. *2011 IEEE 19th International Conference on Program Comprehension*, 31–40. https://doi.org/10.1109/ICPC.2011.35