International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys. 55, No.55 (Mon-2024)

http://dx.doi.org/10.12785/ijcds/XXXXXX

ScaMaha: A Tool for Parsing, Analyzing, and Visualizing Object-Oriented Software Systems

Ra'Fat Al-Msie'deen[1]

[1]*Department of Software Engineering, Faculty of IT, Mutah University, Mutah 61710, Karak, Jordan*

Received 55 Mon. 2024, Revised 55 Mon. 2024, Accepted 55 Mon. 2024, Published 88 Mon. 2024

Abstract: Reverse engineering tools are required to handle the complexity of software products and the unique requirements of many different tasks, like software analysis and visualization. Thus, reverse engineering tools should adapt to a variety of cases. Static Code Analysis (SCA) is a technique for analyzing and exploring software source code without running it. Manual review of software source code puts additional effort on software developers and is a tedious, error-prone, and costly job. This paper proposes an original approach (called ScaMaha) for Object-Oriented (OO) source code analysis and visualization based on SCA. ScaMaha is a modular, flexible, and extensible reverse engineering tool. ScaMaha revolves around a new meta-model and a new code parser, analyzer, and visualizer. The ScaMaha parser extracts software source code based on the Abstract Syntax Tree (AST) and stores this code as a code file. The code file includes all software code identifiers, relations, and structural information. The ScaMaha analyzer studies and exploits the code files to generate useful information regarding software source code. The software metrics file gives unique metrics regarding software systems, such as the number of method access relations. Software source code visualization plays an important role in software comprehension. Thus, the ScaMaha visualizer exploits code files to visualize different aspects of software source code. The visualizer generates unique graphs about software source code, like the visualization of inheritance relations. ScaMaha tool was applied to several case studies from small to large software systems, such as drawing shapes, mobile photo, health watcher, rhino, and ArgoUML. Results show the scalability, performance, soundness, and accuracy of ScaMaha tool. Evaluation metrics, like precision and recall, demonstrate the accuracy of ScaMaha in parsing, analyzing, and visualizing software source code, as all code artifacts were correctly extracted.

Keywords: Software engineering, Reverse engineering, Software re-engineering, Object-Oriented source code, Static code analysis, Software visualization, ScaMaha tool.

1. Introduction

Reverse engineering tools are necessary to cope with the complexity of software systems. Also, such tools should cope with the specific requirements of the various reverse engineering tasks, like software comprehension and visualization. Thus, these tools should adapt to a wide range of cases [1]. To analyze software code, tools need to represent it. Such a representation should be comprehensive. Software comprehension is still a manual activity. Thus, advanced tools will help developers fully understand complex systems [2]. This paper presents the ScaMaha tool, which is a reverse engineering tool for performing software analysis and visualization. The core of the ScaMaha tool revolves around the meta-model, code parser, code analyzer, and code visualizer. With ScaMaha, tool developers can analyze and visualize software code. Also, developers can develop new, specific, and dedicated reverse engineering tools based on the infrastructure of the ScaMaha tool.

The main challenge in evolving and maintaining legacy software products is comprehending the chosen software. Reverse engineering is the process of studying and analyzing software products [3]. The goal of this process is to identify the product's components and their relationships. Furthermore, this process aims at creating several representations of the software product in another shape or at a different level of abstraction. The main goal of the software re-engineering process is to improve or modify current software so it can be comprehended, administered, and used again as new software [4], [5].

Figure 1 illustrates the typical organization of a re-engineering environment. The left side of Figure 1 displays the software source code, which can be brought into this environment using suitable code parsers, such as ScaMaha parser. Also, Figure 1 displays the main repository of this environment, which holds the software code (*aka.* code database). The repository includes an abstracted model of the software code, which is based on ScaMaha's meta-model. The right side of Figure 1 displays the tools (*e.g.,* ScaMaha analyzer and visualizer) that utilize the repository as their information source to perform specific tasks. The key part that makes all tools work together is the repository's meta-model.

To support software comprehension, visualization, and maintenance, meta-models are often employed during software reverse engineering tasks to describe the components of software and their relationships. Reverse engineering tools frequently define their own meta-models depending on the intended goals and functionalities [6].

Each programming language (*e.g.,* Java) has its own rigorous syntax that can be thought of as a collection of predetermined rules revealing all probable programming language constructions. Analyzing the raw software code with these predefined rules allows depicting it in the shape of a parse tree and, after that, as an AST [7]. By dealing with this structure, scholars enhance their findings for several software engineering duties, such as code visualization and comprehension [8]. In SE domain, tools are typically based on several tools executing particular tasks, such as mining code identifiers, performing code analysis, and visualizing code [9]. Thus, a common code meta-model (*cf.* Figure 1) is required to represent information or facts about the software that is being analyzed [10].

This study presents ScaMaha tool, which parses, analyzes, and visualizes OO source code. The ScaMaha parser generates an XML file (called a code file) representing software code. Software developers can use this code parser in any work that deals with software code, such as feature identification [11], [12], [13] and software evolution [14]. ScaMaha tool extracts all code identifiers and relations.

In addition to code parser, this study presents the ScaMaha analyzer. This analyzer accepts as input the software code file that was produced by ScaMaha parser to generate a software metrics file (an XML file). The software metrics file contains quantitative information regarding software source code, like the number of software classes and methods. Software engineers can use or extend the current version of ScaMaha analyzer to examine other aspects of software code. The novelty of this analyzer is that it exploits code information to uniquely identify some features of software code, such as the number of method invocations and attribute accesses. ScaMaha analyzer can be used in several software engineering research areas, such as software maintenance.

In addition to a code parser and analyzer, ScaMaha also presents a code visualizer. This visualizer accepts as input the code file generated via ScaMaha parser and produces a set of code graphs. Each graph addresses a specific aspect of software code. For example, one kind of graph shows the structural information of software code, and another one shows the inheritance relations across software classes. Software visualization is a hot topic in the software engineering domain [15]. Graphs give software developers an indication of software size (or complexity level). In addition, graphs present code information in its simplest form to software developers. The current version of ScaMaha visualizer can be easily extended to include other kinds of graphs covering different aspects of software code.

In this study, the only input for ScaMaha is the OO source code. The first step is aimed at parsing software source code statically using the AST. The output produced by ScaMaha code parser (*i.e.,* parsed code) can be named as metadata (*i.e.,* an XML file). The parsed code is saved into code database for further use. The second step of ScaMaha tool is aimed at analyzing the software source code. The output produced via ScaMaha code analyzer can be named as metadata (*i.e.,* an XML file). The analyzed code (*i.e.,* software metrics) is kept in the tool database. Finally, the third step of the suggested tool is aimed at visualizing the software source code. Several graphs regarding software code are produced and stored in the tool database. Figure 2 briefly shows the use of ScaMaha tool in this work.

Figure 3 presents the use-case diagram of ScaMaha tool. The use-case diagram shows all possible interactions between external users (*i.e.,* software engineers) and ScaMaha. The use-case diagram displays a collection of actions (called use-cases) that are supported by the proposed tool, such as parse and visualize software source code.

This paper proposes an automatic approach to analyzing and visualizing OO software systems. Figure 4 illustrates the main elements of the ScaMaha approach.

2. Related Work

This section offers a literature review associated with ScaMaha contributions. The closest works to ScaMaha are chosen and offered in this section.

Bruneliere *et al.* [16] suggested a semantic and syntax analysis based parser for the creation of AST and metrics for multi-language software systems. Their meta-modeling tool [17] is utilized to analyze multi-language applications [18].

VerveineJ is a parser developed in Java that constructs an MSE file from software source code [19]. Based on Eclipse Java Development Tools (JDT), VerveineJ parses software code written in Java to export it in the MSE format that is utilized by the Moose data analysis platform [20], [21]. Moreover, VerveineJ is used to extract relationships (or dependencies) from software source code.

Janes *et al.* [22] proposed an uncommon open-source solution that prevents producing parsers from scratch or dealing with parser generators. They proposed and described how to employ parsers included in the Eclipse Integrated Development Environment (IDE) [23] to parse

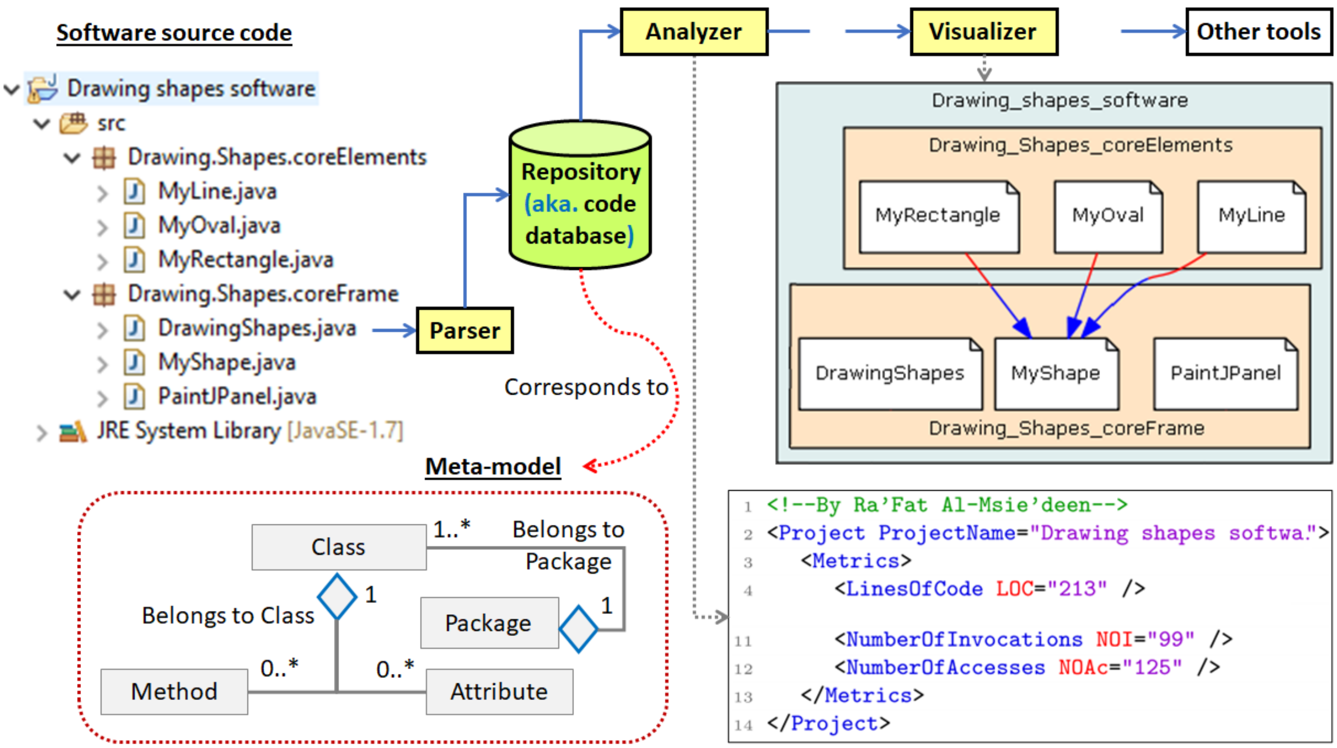


Figure 1. Typical infrastructure for re-engineering tools.

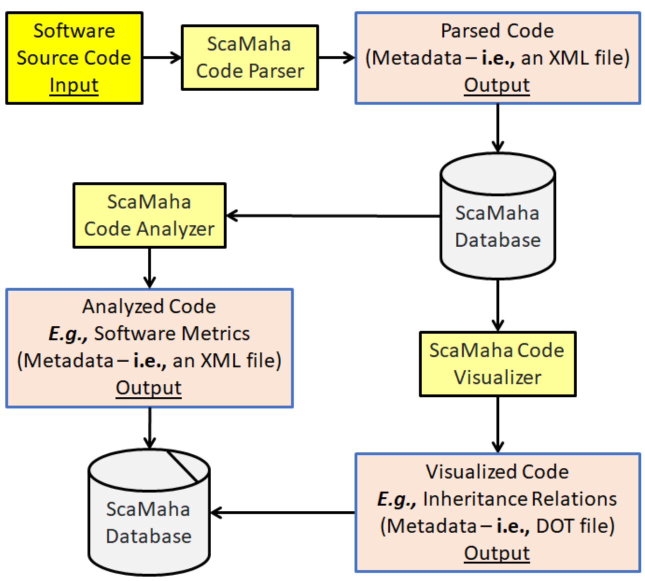


Figure 2. An overview of ScaMaha tool.

software code, such as the JSDT parser [24].

Parsing and analyzing software source code is a critical part of the reverse engineering process. Nowadays, several source code parsers exist. Software engineers can utilize those parsers to deal with numerous software engineering activities, like software comprehension and visualization. O'Hara and Slavin [25] described in their work a collection of tools for parsing and analyzing many different programming languages, involving legacy languages like Fortran. Table I displays the main characteristics of ScaMaha tool.

Wettel and Lanza [26] have suggested an automatic tool called CodeCity. This tool visualizes software source code as a city metaphor. CodeCity is an interactive, three-dimensional software visualization tool [27]. CodeCity shows the software code as a city, where the buildings (*resp.* districts) of the city represent software classes (*resp.* packages). In CodeCity, building dimensions display values of software metrics, like the number of methods or the number of attributes [28]. While this study presents an automatic tool called ScaMaha, which aims at parsing, analyzing, and visualizing software source code. ScaMaha visualizer generates several graphs regarding several aspects of software code, such as code organization and relations.

The code parser is used in several feature identification (*aka.* feature location) studies, such as in [11], [12], [13]. It has been used to extract the main source code elements (*e.g.,* packages and classes) and relations (*e.g.,* attribute access and method invocation) from software products. Also, code parsers are utilized to construct the feature model [29], [30] from the OO source code of a collection of software product variants, such as in [31], [32], [33], [34], [35]. Moreover, code parser are exploited to visualize all software identifier names as tag clouds (*aka.* word clouds), such as in [36], [37], [38], [39]. Furthermore, parsers are

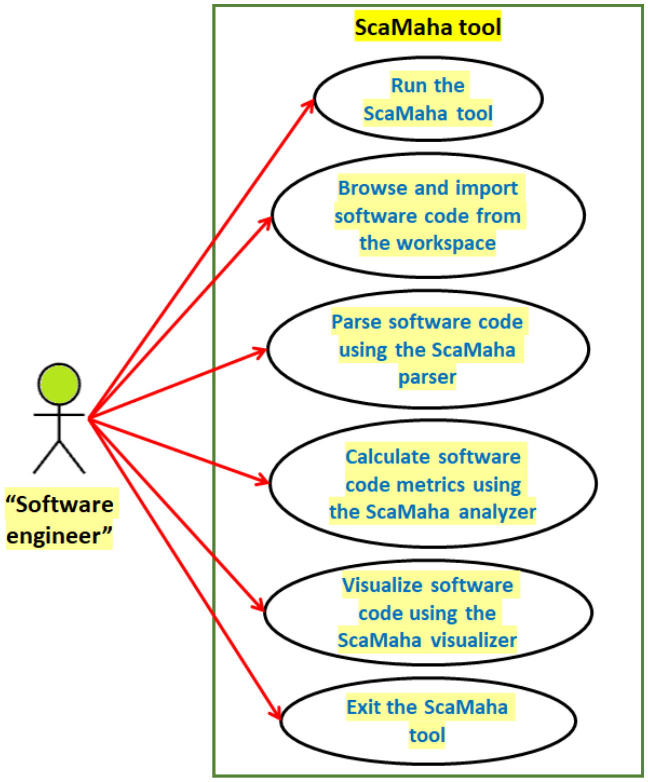


Figure 3. The use-case diagram of ScaMaha tool.

employed to identify the traceability links between software source code and its artifacts (*e.g.,* software requirements [40], [41]), such as in [42], [43].

Code parsers are utilized to study OO software evolution based on software identifiers and code relations, such as in [44], [14]. Furthermore, code parsers are exploited in OO source code summarization studies, such as in [45]. Moreover, code parsers are utilized in several studies concerning software source code documentation, such as in [46], [8], [47]. For more information about those parsers, interested readers can refer to the published articles for more details. All these studies show the value of code parsers in the reverse engineering (*resp.* re-engineering) process.

The Java language has a wide variety of parsers due to its long history of development, popularity, and huge number of applications nowadays. Numerous tools exist that turn software code into a tree-like structure, such as interpreters and compilers. There are several Java parsers for various contexts because there are so many different Java applications, such as Spoon [48], SrcML [49], and SuperParser [7].

The methods of software source code visualization have become increasingly utilized to support software engineers in software understanding [50]. In software visualization, some methods aim at displaying software source code in a recognized environment, like a forest [51] or a city [28]. Another method is to generate what is called a polymetric view [52], described as a lightweight visualization technique supplemented with several metrics regarding software code [15]. ScaMaha visualizes different aspects of software source code, such as code organization and relations.

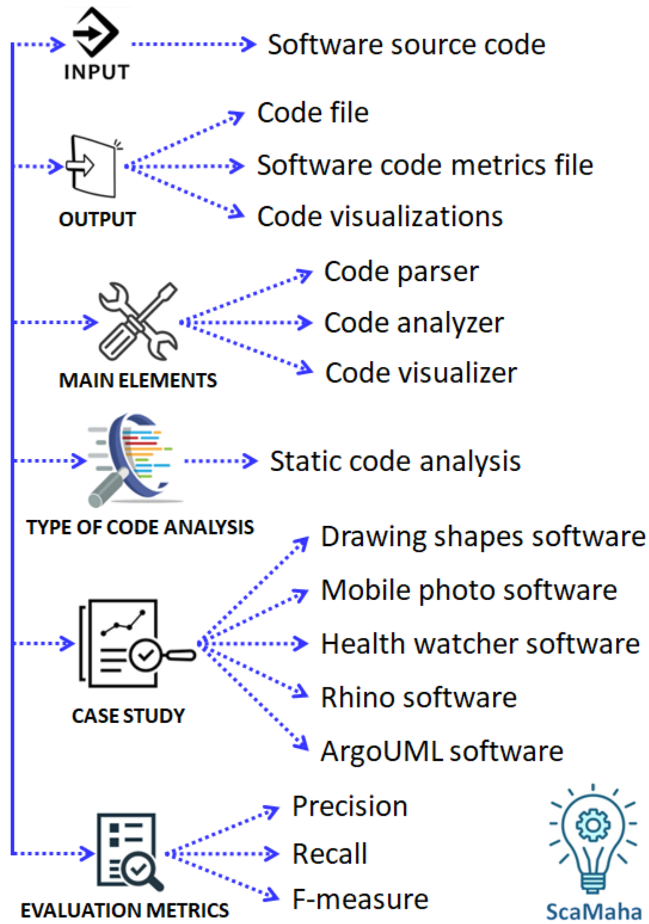


Figure 4. The main elements of the ScaMaha approach.

Specialized reverse engineering tools are important and needed these days. Reverse engineering tools are required to deal with the complexity of products and the particular requirements of various tasks, like software comprehension and reuse [53]. Thus, reverse engineering tools should fit a wide range of circumstances. ScaMaha is a reverse engineering tool for parsing, analyzing, and visualizing software source code. Moose is a well-known reverse engineering tool [54]. It began as a research project around 24 years ago. MODMOOSE is the new version of Moose [1]. Tool developers can develop specialized reverse engineering tools with MODMOOSE. Moose was based on the Famix meta-model [10]. MODMOOSE uses FamixNG (a composable meta-model of programming languages), where FamixNG is a redesign of Famix. MODMOOSE utilizes Roassal to script and display interactive graphs [55]. Roassal is

TABLE I. ScaMaha tool characteristics.

ScaMaha code parser									
Code entities (or identifiers)									
	Class				Method				
Package	Comment	Super-class	Interface	Attribute	Access level	Comment	Parameter	Local variables	Exception
×	×	×	×	×	×	×	×	×	×

Code relations (or dependencies)	
Inheritance	✓
Attribute access	✓
Method invocation	✓

ScaMaha code analyzer	
Code metrics	
Lines of code	×
Number of packages	×
Number of classes	×
Number of attributes	×
Number of methods	×
Number of comments	×
Number of local variables	×
Number of inheritance relations	×
Number of attribute access relations	×
Number of method invocation relations	×

ScaMaha code visualizer	
Code visualizations	
Code organization	✓
Class inheritance relations	✓
Method invocation relations	✓
Polymetric view	✓

a visualization engine in MODMOOSE. MODMOOSE mainly exploits Roassal to show code entities and their relations in several forms or colors. MODMOOSE uses the MSE file format to describe the source code models [10]. Where the MSE has been utilized to save FamixNG models. Thus, a software engineer uses an external parser to generate the MSE file of software code, then loads the MSE file into MODMOOSE in order to analyze or visualize it. MODMOOSE is the closest tool to the ScaMaha tool.

Lyons *et al.* [56] have suggested the lightweight multilingual software analysis method to analyze software systems. They use several code parsers, one for every programming language. The main goal of this work is to create a software engineering tool that addresses large and complex software systems in a lightweight and extensible style. The current version of the ScaMaha tool uses only one code parser for the Java language.

A study of current approaches confirmed the need to offer a comprehensive tool in order to analyze and visualize outdated OO software systems. This work suggests the ScaMaha tool, which uses SCA to perform several activities on software code, like code visualization and analysis. This tool accepts only software code and produces a set of code artifacts, which are the code file, code metrics file, and code visualizations. Moreover, this tool helps software engineers understand and maintain legacy software systems. Also, software engineers can easily extend the current version of the tool in order to include other functionalities.

3. Analyzing and Visualizing OO Source Code via ScaMaha Tool

In this study, the author used Java as a target programming language due to its wide adoption in the software engineering field. The Java language is a popular target for both semantic and syntactic code analysis, as well as studies on code visualization [8], code summarization [45], feature location [11], [12], and re-engineering of software product variants into a Software Product Line (SPL) [31], [32], [57], [58]. Application Programming Interfaces (APIs) are available to access and manipulate Java code through Eclipse development tools. Eclipse JDT project provides access to Java code through AST and the Java model. The Java model is displayed as a tree-like structure, and it is used internally to represent each Java project. AST is a comprehensive tree representation of software code.

This study suggests an automatic approach to parse, analyze, and visualize the OO software system in a unique manner. The main contribution of this work to the software engineering field is to suggest ScaMaha tool. ScaMaha accepts as input just software source code and produces as output a set of software artifacts, like the code file, software metrics file, and code graphs. ScaMaha tool bases itself on static analysis of software source code. ScaMaha parser extracts all code identifiers and relations. Then, ScaMaha analyzer identifies comprehensive software code metrics, and, at last, ScaMaha visualizer gives a set of graphs for different aspects of software code. Figure 5 gives an overview of ScaMaha tool.

ScaMaha operates on a model of software source code, namely the ScaMaha model. To analyze the OO software system, you must first create a model of it using ScaMaha. Figure 6 shows the core of the ScaMaha meta-model. This meta model shows the main OO software identifiers, like software packages, classes, attributes, and methods. Moreover, this meta model displays the main OO software relationships, such as inheritance, method invocation, and attribute access. The core of ScaMaha involves a language-independent meta-model that can show several OO languages in a uniform style. In most cases, the developer gets sufficient information if he explores the basic types of entities that model an OO software system. These are code identifiers (*e.g.,* package) and the relationships (*e.g.,* inheritance) between them.

This model shows the majority of OO entities. For

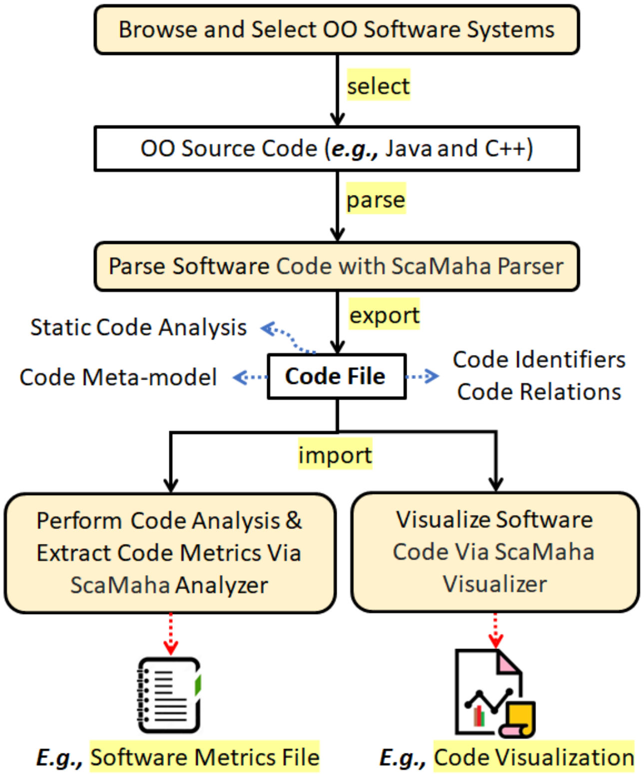


Figure 5. Analyzing and visualizing OO source code with ScaMaha tool.

instance, it shows that a method has parameters, comments, and local variables. However, while this model doesn't show all OO entities, it is also valuable since, for most practical purposes in the reverse engineering (*resp.* re-engineering) process, it is all software developers need. Also, the ScaMaha model shows that a class (*resp.* package) has methods (*resp.* classes), and a method (*resp.* class) belongs to a class (*resp.* package). Furthermore, this model shows that an inheritance relation denotes a connection between two classes (*i.e.,* the subclass and the superclass). Moreover, the ScaMaha model shows that an attribute access relation denotes a connection between method and attribute. Also, it shows that a method invocation relation means a connection between one method and another method.

The first step of code analysis is to explore the software directory (or any workspace) to select the OO software that the developers want to analyze. Thus, let's say there is a directory (repository) of OO software projects that developers need to analyze (*e.g.,* the source code from the Eclipse workspace). In this case, developers will browse the directory to select a particular software project in order to get a copy of its source code (*cf.* Figure 5). If software developers want to analyze Java software, they can use any directory, such as a git repository (*i.e.,* GitHub) [59].

In the second step of SCAMaha, developers parse OO source code to build the SCAMaha model (*cf.* Figure 6). Once developers have a copy of the software source code, they can create a SCAMaha model of software code using the SCAMaha parser. To parse the OO source code, SCAMaha depends on the static code parser. In this study, the most important entities of source code are considered and parsed, such as packages, classes, methods, and attributes. Also, key relations between main code entities (inheritance, invocation, and access) are considered and parsed. SCAMaha parser generates an XML file of software source code.

In this work, XML is a generic file format that represents the ScaMaha code model. Thus, ScaMaha parser converts OO source code to an XML file format. ScaMaha parser produces an XML file for each software product. This file includes all code entities (or identifiers) and relationships (or dependencies) between those entities. Listing 1 shows the XML format corresponding to the ScaMaha meta-model.

As an illustrative example, this study considers the mobile photo software system. ScaMaha used this software to better explain its work. Mobile photo is open-source software that allows users to manipulate photos on their mobile devices [60]. This study considers the first release of mobile photo software [61].

ScaMaha exploits Eclipse JDT and AST to parse software systems written in Java. Several studies utilized Eclipse AST to access, read, and manipulate the software code. XML enables easy interchange of metadata between several tools in diverse environments (*cf.* Figure 2). Moreover, XML is a human-readable format. The XML structure matches the author's requirements for the representation and description of the ScaMaha model.

The chosen way to load a model in ScaMaha is through an XML file. XML is a compact, simple, and robust format. This study exploits XML to represent the core of the ScaMaha model. In order to analyze OO source code, software developers need to load the code model as an XML file into the ScaMaha tool. The ScaMaha analyzer aims at analyzing software code and extracting useful software code metrics. Developers can extend the current work of the ScaMaha analyzer by performing other analysis activities on software code. Software developers can use the ScaMaha analyzer to obtain several code metrics. For instance, the software LOC and the number of packages. The mined code metrics file gives the software engineer an indication about the size (complexity level) of the software system. In this study, the ScaMaha analyzer considers the software metrics presented in Table II. Analyzer of ScaMaha can easily extend to include other code metrics.

Source code metrics are measurements utilized to characterize software code. A code metric is a useful quantitative measure extracted from software's source code. Size is the most recognizable metric for software source code.

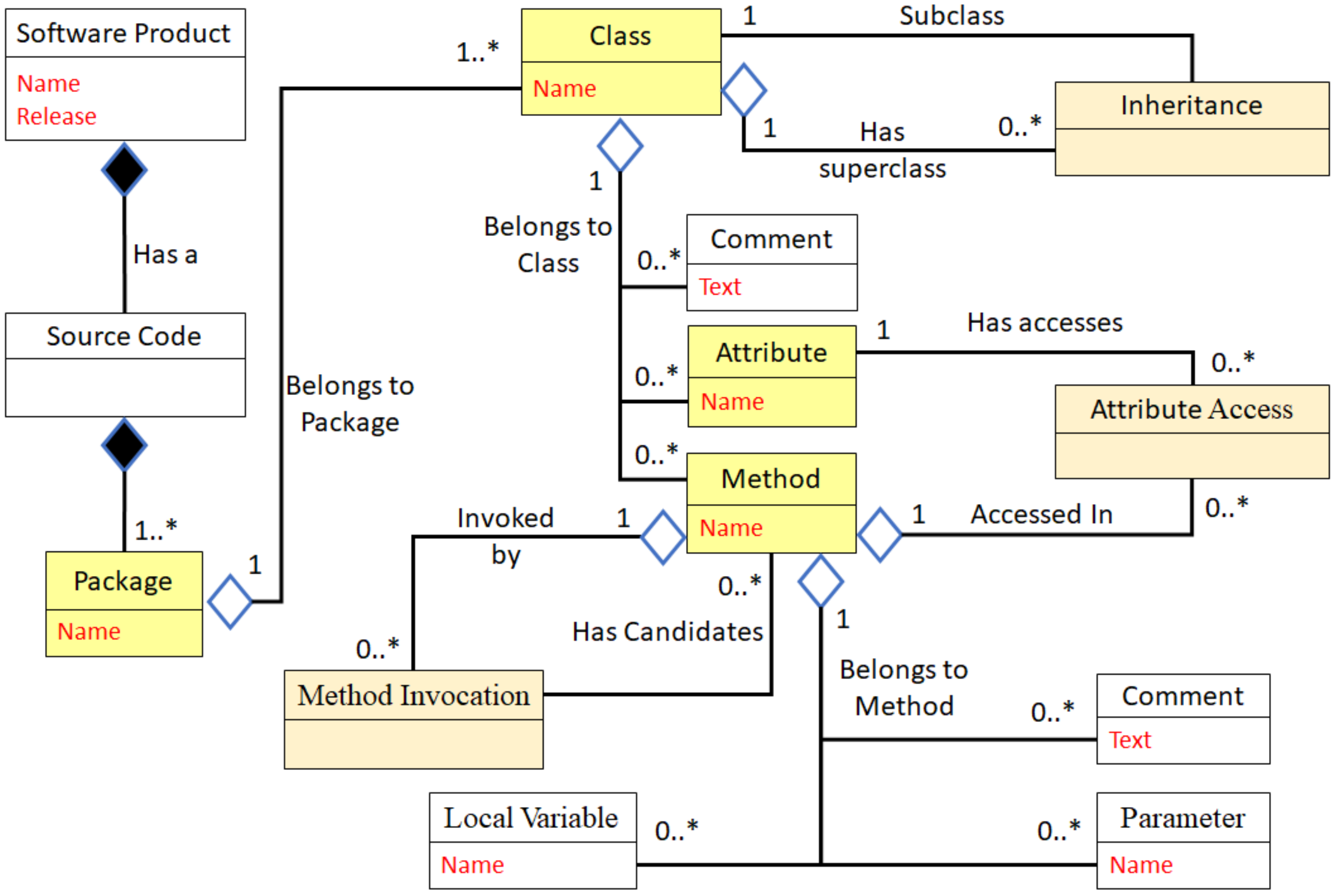


Figure 6. The core of the ScaMaha meta-model.

TABLE II. Software code metrics of the ScaMaha code analyzer.

Metric	Abbreviation
Lines of Code	LOC
Number of Packages	NOP
Number of Classes	NOC
Number of Attributes	NOA
Number of Methods	NOM
Number of Comments	NOCo
Number of local variables	NOLv
Number of inheritance relations	NOIn
Number of attribute access relations	NOAc
Number of method invocation relations	NOI

The number of LOC is the easiest method of measuring software size. All code metrics given in Table II are static code metrics. Static code metrics are metrics obtained directly from software source code, like the LOC metric. A subgroup of static code metrics are OO code metrics, as they are also metrics obtained from software code itself, such as NOIn, NOI, and NOAc metrics. Listing 2 shows the extracted software code metrics from mobile photo software by the ScaMaha analyzer.

```
<!--By Ra'Fat Al-Msie'deen-->
<Project ProjectName="Mobile photo software">
  <Metrics>
    <LinesOfCode LOC="1229" />
    <NumberOfPackages NOP="10" />
    <NumberOfClasses NOC="15" />
    <NumberOfAttributes NOA="56" />
    <NumberOfMethods NOM="91" />
    <NumberOfComments NOCo="250" />
    <NumberOfInheritances NOIn="14" />
    <NumberOfInvocations NOI="298" />
    <NumberOfAccesses NOAc="631" />
  </Metrics>
</Project>
```

Listing 2. Software code metrics for the mobile photo software.

In this work, the use of visualization speeds up the comprehension of legacy OO source code. By utilizing ScaMaha to analyze and visualize software source code, software developers are able to speed their maintenance, reuse, and comprehension of software products. Source code visualization aims at producing graphical representations (or annotations) of software code in order to help comprehend and analyze it. In the software engineering domain, the code visualization process plays an important role in understanding how large software products work [8].

```xml
<!--By Ra'Fat Al-Msie'deen-->
<Project Name="-">
   <Packages>
      <Package Name="-">
         <Classes>
            <Class Name="-" AccessLevel="-" isInterface="-" Superclass="-">
               <SuperInterfaces>
                  <Interface InterfaceName="-" />
               </SuperInterfaces>
               <Comments>
                  <Comment CommentText="-" />
               </Comments>
               <Attributes>
                  <Attribute Name="-" AccessLevel="-" Type="-" isStatic="-" />
               </Attributes>
               <Methods>
                  <Method Name="-" AccessLevel="-" ReturnType="-" isStatic="-">
                     <Parameters NumberOfParameters="-">
                        <Parameter Name="-" Type="-" />
                     </Parameters>
                     <Comments>
                        <Comment CommentText="-" />
                     </Comments>
                     <LocalVariables>
                        <LocalVariable Name="-" Type="-" />
                     </LocalVariables>
                     <AttributeAccesses>
                        <Access Name="-" Type="-" HowIsItUsed="-" />
                     </AttributeAccesses>
                     <MethodInvocations>
                        <MethodInvocation Name="-" Arguments="-" />
                     </MethodInvocations>
                     <MethodAssignments>
                        <Assignment LeftHandSide="-" RightHandSide="-" />
                     </MethodAssignments>
                     <MethodExceptions>
                        <Exception ExceptionType="-" />
                     </MethodExceptions>
                  </Method>
               </Methods>
            </Class>
         </Classes>
      </Package>
   </Packages>
</Project>
```

Listing 1. XML format corresponding to the ScaMaha meta-model.

The main goal of the ScaMaha visualizer is to visualize software code entities and relations. All code entities and relations are defined in the ScaMaha core meta-model (*cf.* Figure 6). To visualize software code via the ScaMaha visualizer, software developers need to load code files using ScaMaha's importer. Then, the suggested tool will generate different visualizations covering several aspects of software code. ScaMaha exporter stores all code visualizations in the tool workspace. Thus, software developers can explore ScaMaha's workspace to see all the code visualizations. Software developers study code visualizations in order to analyze and understand software products. Figure 7 briefly shows the core parts of the ScaMaha tool.

ScaMaha visualizes several aspects of software code. The visualization of code organization shows the main code entities as boxes. Code organization visualization represents software packages, classes, and methods. Developers can easily extend the current visualization by adding other entities of code, like software attributes. In code organization visualization, the big box represents the whole software code, while other boxes represent main code entities. This visualization shows that the box may contain other boxes. For instance, the package box includes all classes belonging to this package. Also, the class box includes all methods belonging to this class. The main objective of this visualization is to show the organization of code in terms of packages, classes, and methods. Figure 8 shows the code organization visualization of mobile photo software.

Also, code organization visualization provides software engineers with structural information about software code. Structural information is the way developers arrange and group software code elements such as package, class, meth-

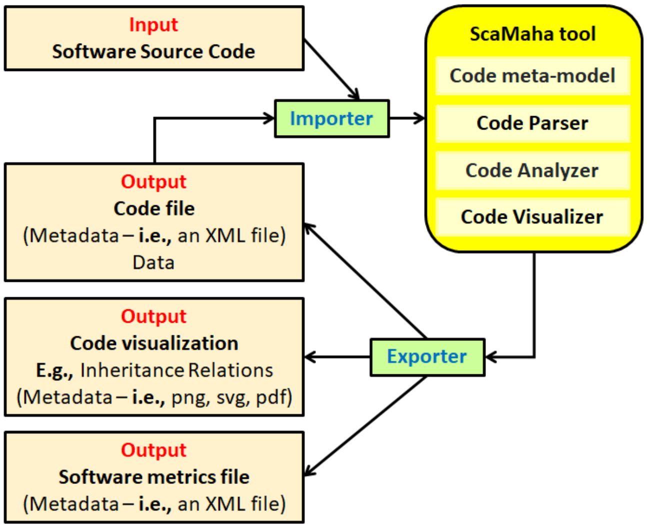


Figure 7. An overview of the core parts of the ScaMaha tool.

ods, and attributes. In this work, well-structured information about code entities makes software code easier to understand by software engineers.

ScaMaha also generates a polymetric view of software source code. This view depends on software packages. Where it represents each software package as a box including a set of package metrics. The name of each package is placed on the top of the box. ScaMaha approach considers the following metrics for each package: LOC, NOC, NOA, NOM, NOCo, NOLv, NOIn, NOI, and NOAc. Figure 9 shows a polymetric view of mobile photo software.

In this work, ScaMaha visualizes inheritance relations between software classes. The main goal of inheritance relationships is to minimize code complexity and size. The inheritance relationship gives an indication of the strong connection between software classes. The visualization of class inheritance relations gives important information about legacy (or outdated) code. This kind of visualization helps software developers when they want to analyze, reuse, understand, and maintain existing code. Figure 10 shows the ScaMaha visualization of class inheritance relations from mobile photo software.

In the OO language, a method is a function defined inside a class. Thus, a software class may contain several methods. Method communicates with other methods in software via method invocation relations. So, method invocation occurs between software methods when a method calls (or invokes) other methods. Thus, a particular method may invoke other methods of the same class or of different classes (*cf.* Figure 11). Usually, a method calls another method by using its name and arguments. Also, methods invoke other methods in order to achieve specific functionality. This study considers method invocation as an important code relationship. ScaMaha visualizes invocation relations across

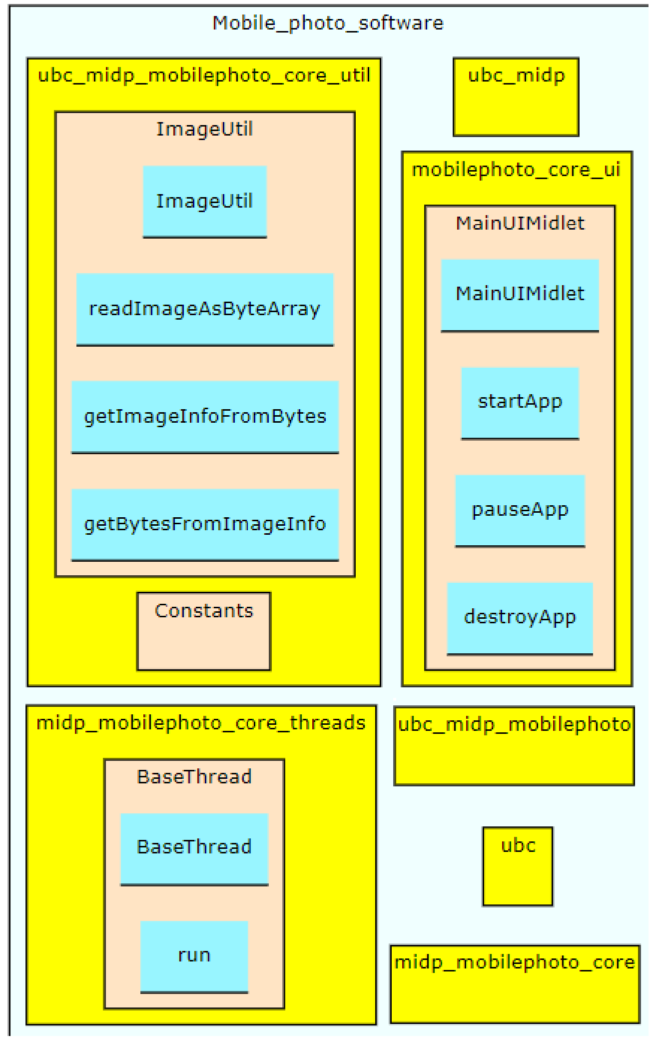


Figure 8. Code organization visualization of mobile photo software (partial) [62].

software methods in a perfect way. Software developers explore the extracted visualization in order to comprehend the software code. Figure 11 shows the ScaMaha visualization of method invocation relations from mobile photo software.

The ScaMaha approach evaluates the produced results using precision, recall, and F-measure metrics [63]. For source code identifiers and relations, the precision metric is the percentage of correctly recovered identifiers (*resp.* relations) to the total number of recovered identifiers (*resp.* relations). While the recall metric is the percentage of correctly recovered identifiers (*resp.* relations) to the total number of relevant identifiers (*resp.* relations). Finally, the F-measure metric quantifies a trade-off among precision and recall metrics; thus, it provides a high value just in cases where both recall and precision metrics are high.

All evaluation metrics of ScaMaha have values between zero (0%) and one (100%). If the recall metric is equal to

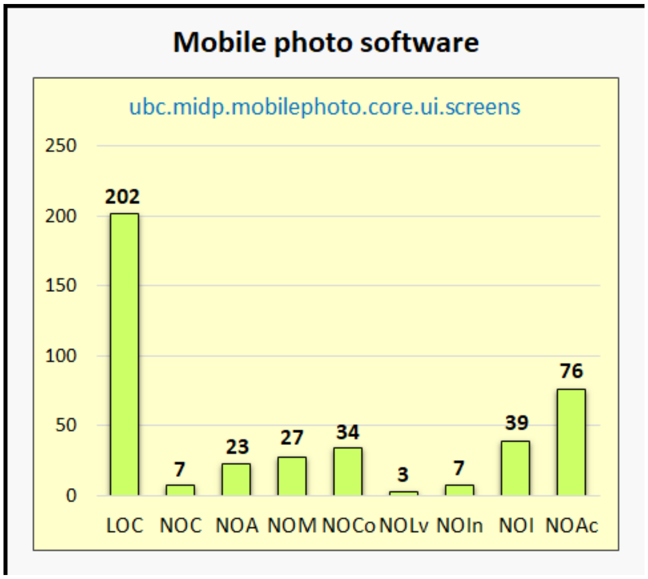


Figure 9. Polymetric view of mobile photo software based on its packages. This view uses the following metrics for each package: LOC, NOC, NOA, NOM, NOCo, NOLv, NOIn, NOI, and NOAc (partial) [62].

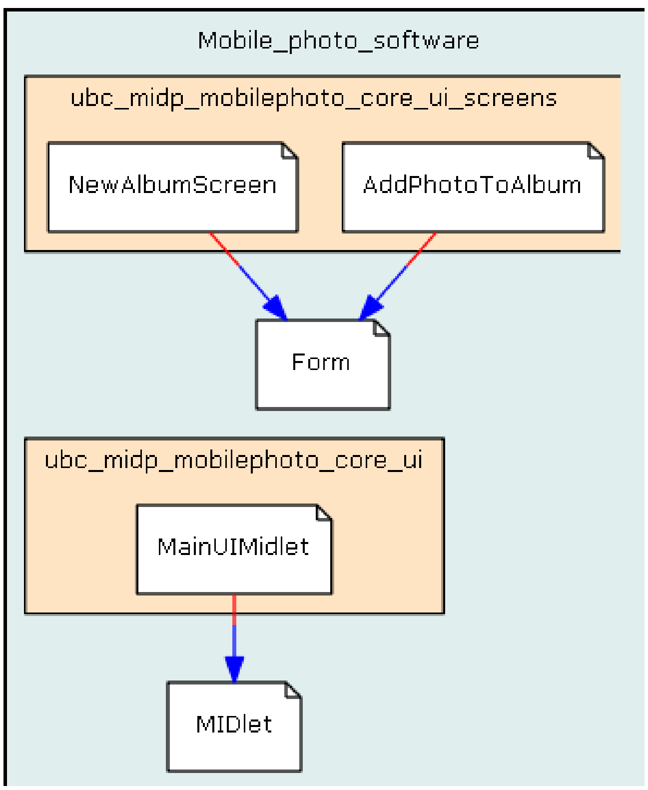


Figure 10. Visualization of class inheritance relations for the mobile photo software (partial) [62].

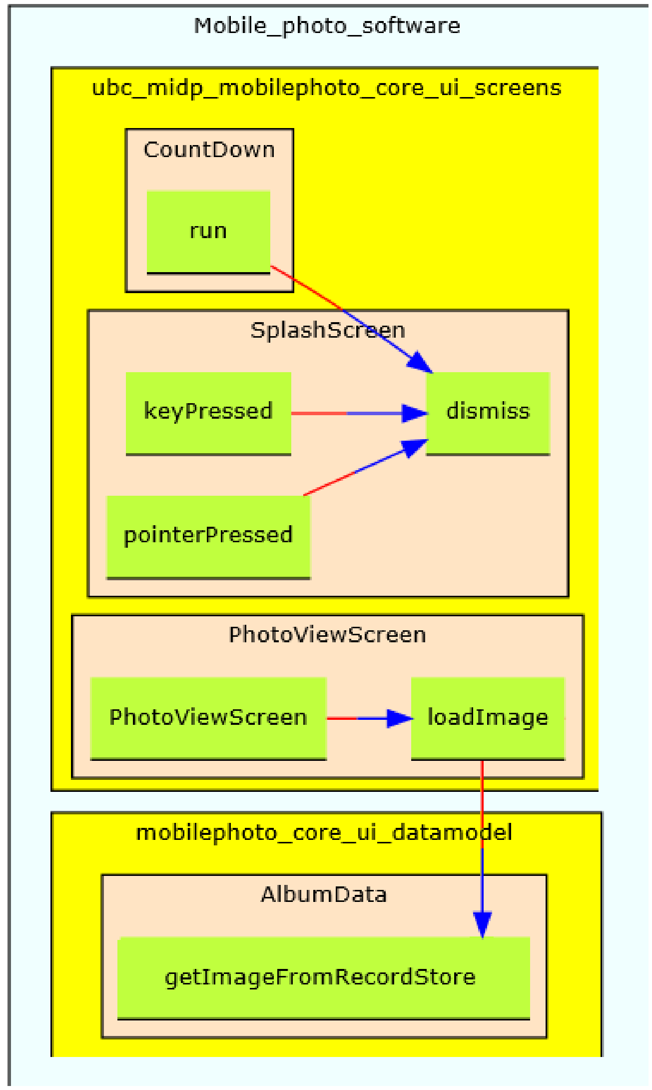


Figure 11. Visualization of method invocation relations for the mobile photo software (partial) [62].

100%, this means that all relevant identifiers (or relations) are recovered. If the precision metric is equal to 100%, this means that all retrieved identifiers (or relations) are relevant. If the F-measure metric is equal to 100%, this means that both recall and precision are high (100%) [31].

In mobile photo software, the proposed approach returns 15 classes from the software code. In this case, the values of all evaluation metrics are equal to 100% since the software code actually contains only 15 classes. Furthermore, for the method invocation relations, the ScaMaha returns 298 relations from the software code. In this circumstance, the values of all evaluation metrics are equal to 100% since the software code actually contains 298 invocation relations. For software code metrics, the retrieved value of each metric is 100% correct. Thus, the values of all evaluation metrics are equal to 100%. For code visualizations, all types of code visualizations have high accuracy. For instance, in mobile photo software, the visualization of class inheritance relations obviously shows the 14 inheritance relations. In this case, the values of all evaluation metrics are equal to 100%. Mobile photo is well-documented software [61]. Thus, all code entities, metrics, and relations are known in advance. Therefore, the available software documentation helps in comparing ScaMaha results against it.

4. Experimentation

This section presents the case studies used in this work. Also, it presents and discusses the obtained results. Furthermore, it mentions threats to the validity of the ScaMaha approach.

ScaMaha runs experiments on several software systems, such as drawing shapes [42], mobile photo [60], health watcher [64], Rhino [65], and ArgoUML [66]. Drawing shapes software allows users to draw several kinds of shapes, such as ovals and lines [67]. Mobile photo is introduced and used in Section 3. The health watcher software is a web-based information system that enables users to register complaints about health issues. In this study, the experiment ran on the last version of the health watcher software (*i.e.,* version 10) [68]. Moreover, Rhino is software for JavaScript developed in Java. In this study, the experiment ran on version 1.7R2 of Rhino. While ArgoUML is an open-source software written in Java [69]. It is widely utilized for designing software systems in Unified Modeling Language (UML). It is a large and complex software system (*i.e.,* 271690 LoC). In this work, all experiments are executed on a 2.40 GHz Intel Core i7 PC with 8 GB of RAM. Table III briefly presents the results obtained from all experiments.

In this work, the different sizes of software systems show the scalability of ScaMaha to work with such systems (*i.e.,* small, medium, and large systems). Moreover, all software systems are well documented. Thus, the results of ScaMaha are measurable. In addition, all case studies are well known and employed to assess many approaches in the field of this study (*i.e.,* SCA).

During software maintenance, software engineers consume a considerable amount of time analyzing legacy software system source code in order to understand it. Furthermore, the cost of software maintenance accounts for 50% to 75% of the total cost of the software product [70]. Thus, comprehending software source code is one of the most challenging activities in the maintenance (*resp.* comprehension) of software products.

Usually, software engineers hope to get all code information (*e.g.,* software identifiers and relations) to exploit this information in many software engineering activities (*e.g.,* maintenance, re-engineering, visualization, documentation, and reverse engineering).

The obtained metrics for software code give an indication of software size (or complexity level). A software metrics file gives software engineers rapid information about software code, like the number of software methods.

One of the main contributions of the ScaMaha approach is to give a polymetric view of software packages. In this study, polymetric view is a lightweight visualization method of software source code [52]. Polymetric views supplemented with unique metrics about software source code [50]. Polymetric views assist software engineers in understanding the complexity level of software code in the reverse engineering process. Figure 12 shows a polymetric view of drawing shapes software based on its packages. This view uses the following metrics for each package: LOC, NOC, NOA, NOM, NOCo, NOLv, NOIn, NOI, and NOAc. Figure 12 shows that drawing shapes software consists of two packages. Also, several metrics regarding each package are given in Figure 12.

Software visualization is an important activity in the software engineering domain. Source code visualization is a real implementation of the quote, "A picture is worth a thousand words.". Code visualization gives developers better information compared to textual code information. ScaMaha visualizations are a real reflection of software code. ScaMaha visualizes code identifiers and relations correctly. The visualizer accepts as input the code file and generates as output a collection of code visualizations. Figure 13 shows the ScaMaha visualization of class inheritance relations from the drawing shapes software. Figure 13 gives different views of software code, where it presents structural information (*e.g.,* MyLine class belongs to the coreElements package), identifier names (*e.g.,* MyShape class), and inheritance relations (*e.g.,* MyLine class inherits attributes and methods of MyShape class) from drawing shapes software.

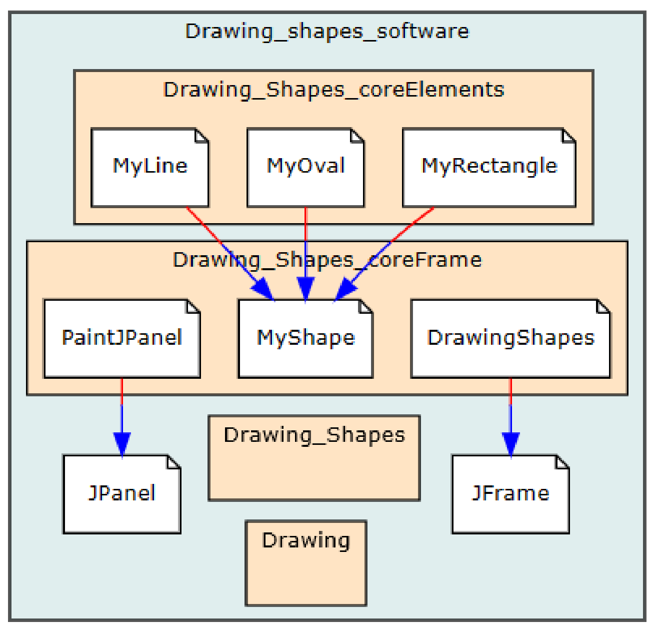


Figure 13. Visualization of class inheritance relations for the drawing shapes software system.

TABLE III. Results obtained from case studies.

ID	Case study	Software size	XML file		Execution time (in ms)	Visualization			
			Code	Metrics		Code[a]	Class[b]	Method[c]	Polyme.[d]
1	Drawing shapes	Small	✓	✓	2095 ms	x	x	x	x
2	Mobile photo	Medium	✓	✓	2850 ms	x	x	x	x
3	Health watcher	Medium	✓	✓	5031 ms	x	x	x	x
4	Rhino	Medium	✓	✓	7965 ms	x	x	x	x
5	ArgoUML	Large	✓	✓	31698 ms	x	x	x	x

[a] Code organization, [b] Class inheritance relations, [c] Method invocation relations, [d] Polymetric view.

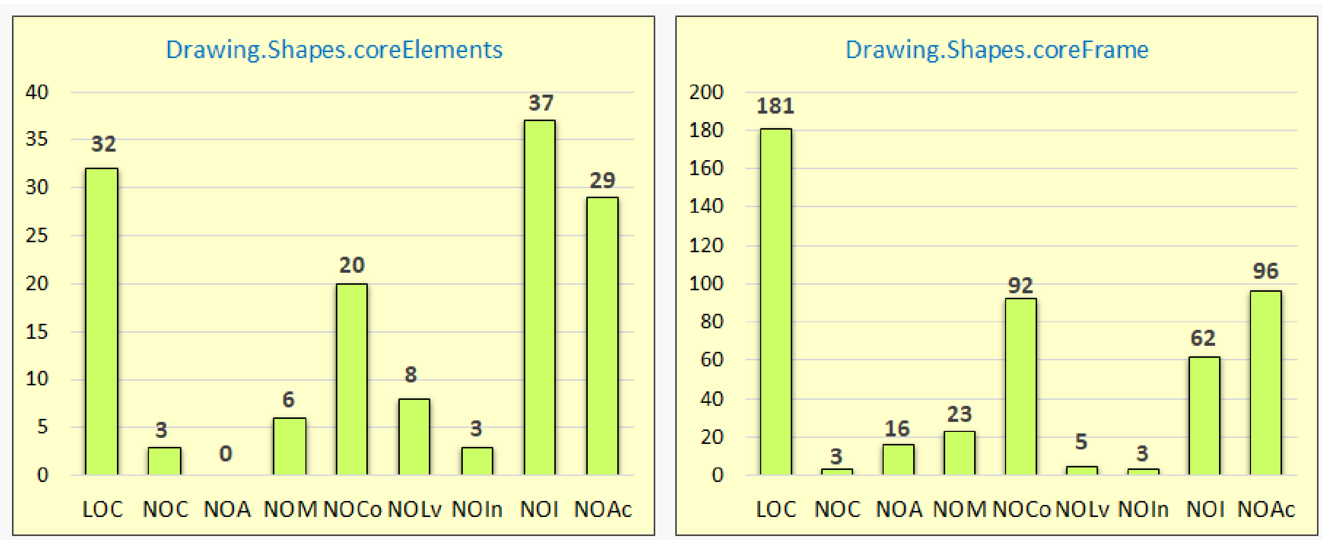


Figure 12. Polymetric view of drawing shapes software based on its packages. This view uses the following metrics for each package: LOC, NOC, NOA, NOM, NOCo, NOLv, NOIn, NOI, and NOAc.

The obtained results show that all evaluation metrics (*i.e.,* precision, recall, and F-measure) appear high (*i.e.,* 100%) for all code entities, relations, and visualizations. This means that all resulted artifacts (*e.g.,* code visualization) from software code are relevant and correct. In this study, the author uses two resources to evaluate the obtained results. The first is the available software documents, and the second is the manual review of software code.

Results show the ability of ScaMaha to retrieve all software identifiers (*e.g.,* software classes and attributes) from any software system. Also, the results show that ScaMaha is able to retrieve all code comments (*i.e.,* class and method comments). Moreover, the ScaMaha tool is capable of retrieving all elements of the method body (*i.e.,* parameter list, local variable, and attribute access). Thus, ScaMaha guarantees that software developers will not lose any code identifier or relation from the software code.

The ScaMaha tool consists of three basic components: a code parser, an analyzer, and a visualizer. Moreover, ScaMaha exploits and reuses two components, which are the JDOM and Graphviz libraries. The ScaMaha component accepts software code as input and produces as output a collection of code artifacts (*e.g.,* the code file and software metrics file). The parser component accepts software code and generates the code file, while the analyzer component accepts the code file and generates a software metrics file. Finally, the visualizer component receives the code file as input and creates several code visualizations. Figure 14 shows an architectural view of the ScaMaha tool in a simplified structure.

Table III shows the results obtained for each case study using the ScaMaha tool. It shows the extracted code file (*resp.* code metrics file) for each case study. Also, it shows the mined code visualizations for each case study. Moreover, the time needed to parse, analyze, and visualize each case study in ms is given in Table III (*i.e.,* execution time). Results show the ability of ScaMaha to work with different software system sizes (*i.e.,* small and large software systems). Also, the results show the ability of the ScaMaha visualizer to provide several visualizations of software code. Thanks to the ScaMaha tool, software developers have parsed, analyzed, and visualized OO software systems in a relatively easy way. All materials and results (*e.g.,* code and metrics files) of ScaMaha experiments are available on the tool's webpage [62].

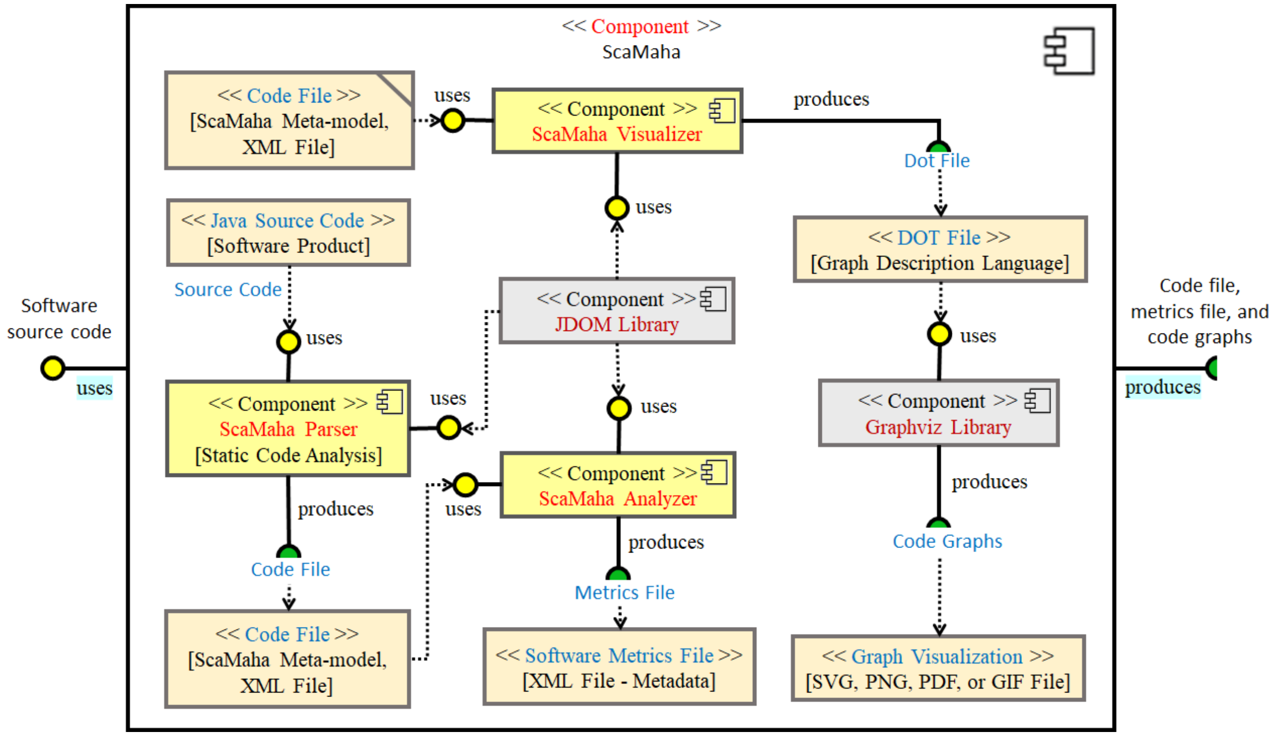


Figure 14. An architectural view of the ScaMaha tool in a simplified structure.

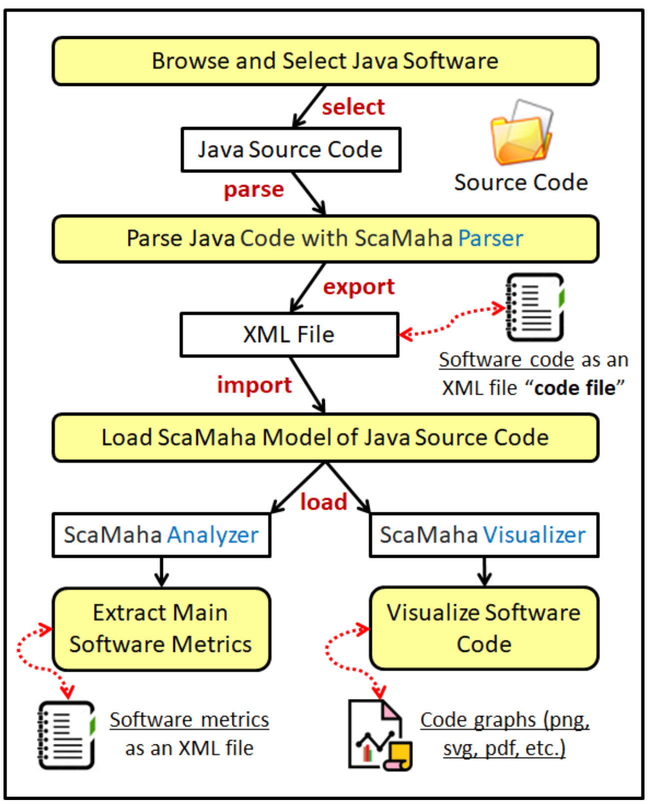


Figure 15. The suggested ScaMaha tool.

5. Conclusion and Future Work

This paper has presented ScaMaha, a tool for parsing, analyzing, and visualizing OO source code like Java. ScaMaha is designed to prevent software engineers from wasting their resources, like effort and time, on manual review of software source code in order to understand it. The novelty of ScaMaha is that it exploits software source code to apply the suggested code analysis and visualization methods in an efficient way to retrieve code information in order to support software engineers in their daily tasks. ScaMaha tool extracts all software code identifiers and relations. Also, it produces a software metrics file that includes statistical information about software source code. On the other hand, it visualizes different aspects of software source code where several graphs are generated, like structural information of software code. ScaMaha extracted several code artifacts from legacy software systems in order to support software engineers in the software comprehension process. The extracted code artifacts of the software code are the code file, the software metrics file, and the code graphs. ScaMaha had been validated and evaluated on several case studies, including drawing shapes, mobile photo, health watcher, rhino, and ArgoUML software. The results of the experiments show the capacity of ScaMaha to recover all software artifacts in an efficient and accurate manner. The evaluation metrics of ScaMaha, like precision and recall, show the accuracy of ScaMaha in parsing, analyzing, and visualizing software source code, as all source code artifacts were correctly obtained.

Regarding ScaMaha's future work, the author plans to extend the current tool by developing a comprehensive tool's parser for all OO languages, like Java and C++. Moreover, additional experimental tests can be performed to verify ScaMaha contributions utilizing open-source and industrial software systems. Also, the author plans to conduct a comprehensive survey regarding current approaches that relate to ScaMaha contributions. Finally, the author of the ScaMaha tool plans to extend the current work (*cf.* Figure 15) by developing a general tool for performing various kinds of analyses and visualizations of any OO software system. The main goal of ScaMaha tool is to assist software engineers in the process of understanding complex and large-sized software systems. Various categories of users, such as scholars in the field of software analysis and tool creators, will exploit ScaMaha tool in their works.

References

[1] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, and M. Derras, "Modular moose: A new generation of software reverse engineering platform," in *Reuse in Emerging Software Engineering Practices*, S. Ben Sassi, S. Ducasse, and H. Mili, Eds. Cham: Springer International Publishing, 2020, pp. 119–134.

[2] W. Pan, X. Du, M. Hua, D. Kim, and Z. Yang, "Identifying key classes for initial software comprehension: Can we do it better?" in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1878–1889. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00160

[3] A. Jain, S. Soner, and A. Gadwal, "Reverse engineering: Journey from code to design," in *2011 3rd International Conference on Electronics Computer Technology*, vol. 5, 2011, pp. 102–106.

[4] W. K. G. Assunção, S. R. Vergilio, and R. E. Lopez-Herrejon, "Reengineering UML class diagram variants into a product line architecture," in *UML-Based Software Product Line Engineering with SMarty*, E. OliveiraJr, Ed. Springer, 2023, pp. 393–414. [Online]. Available: https://doi.org/10.1007/978-3-031-18556-4_18

[5] M. Majthoub, M. H. Qutqut, and Y. Odeh, "Software re-engineering: An overview," in *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, 2018, pp. 266–270.

[6] H. Washizaki, Y.-G. Guéhéneuc, and F. Khomh, "A taxonomy for program metamodels in program reverse engineering," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 44–55.

[7] I. Utkin, E. Spirin, E. Bogomolov, and T. Bryksin, "Evaluating the impact of source code parsers on ML4SE models," *CoRR*, vol. abs/2206.08713, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2206.08713

[8] R. Al-Msie'deen, "Visualizing object-oriented software for understanding and documentation," *International Journal of Computer Science and Information Security (IJCSIS)*, vol. 13, no. 5, pp. 18–27, 2015.

[9] B. Bellay and H. C. Gall, "A comparison of four reverse engineering tools," in *4th Working Conference on Reverse Engineering, WCRE '97, Amsterdam, The Netherlands, October 6-8, 1997*, I. D. Baxter, A. Quilici, and C. Verhoef, Eds. IEEE Computer Society, 1997, pp. 2–11. [Online]. Available: https://doi.org/10.1109/WCRE.1997.624571

[10] S. Ducasse, N. Anquetil, M. U. Bhatti, A. C. Hora, J. Laval, and T. Gîrba, "MSE and FAMIX 3.0: an interexchange format and source code model family," 2011, [Online]. Available: https://inria.hal.science/hal-00646884/document (Accessed: Feb. 5, 2024).

[11] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Feature location in a collection of software product variants using formal concept analysis," in *Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings*, ser. Lecture Notes in Computer Science, J. M. Favaro and M. Morisio, Eds., vol. 7925. Springer, 2013, pp. 302–307.

[12] R. A. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, and S. Vauttier, "Mining features from the object-oriented source code of software variants by combining lexical and structural similarity," in *IEEE 14th International Conference on Information Reuse & Integration, IRI 2013, San Francisco, CA, USA, August 14-16, 2013*. IEEE Computer Society, 2013, pp. 586–593. [Online]. Available: https://doi.org/10.1109/IRI.2013.6642522

[13] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, S. Vauttier, and H. E. Salman, "Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing," in *The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013*. Knowledge Systems Institute Graduate School, 2013, pp. 244–249.

[14] R. Al-Msie'deen and A. Blasi, "The impact of the object-oriented software evolution on software metrics: The iris approach," *Indian Journal of Science and Technology*, vol. 11, no. 8, pp. 1–8, 2018.

[15] P. Lima, J. Melegati, E. Gomes, N. S. Pereira, E. M. Guerra, and P. Meirelles, "CADV: A software visualization approach for code annotations distribution," *Inf. Softw. Technol.*, vol. 154, p. 107089, 2023. [Online]. Available: https://doi.org/10.1016/j.infsof.2022.107089

[16] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "Modisco: a generic and extensible framework for model driven reverse engineering," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 173–174. [Online]. Available: https://doi.org/10.1145/1858996.1859032

[17] H. Bruneliere. (2010) MoDisco. [Online]. Available: https://www.eclipse.org/MoDisco/ (Accessed: Feb. 5, 2024).

[18] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Inf. Softw. Technol.*, vol. 56, no. 8, pp. 1012–1032, 2014. [Online]. Available: https://doi.org/10.1016/j.infsof.2014.04.007

[19] Moose VerveineJ. (2023) VerveineJ. [Online]. Available: https://modularmoose.org/moose-wiki/Developers/Parsers/VerveineJ.html (Accessed: Feb. 5, 2024).

[20] Moose Technology. (2022) VerveineJ. [Online]. Available: https://github.com/moosetechnology/VerveineJ (Accessed: Feb. 5, 2024).

[21] Moose. (2023) MooseTechnology. [Online]. Available: https://modularmoose.org/ (Accessed: Feb. 5, 2024).

[22] A. Janes, D. Piatov, A. Sillitti, and G. Succi, "How to calculate software metrics for multiple languages using open source parsers," in *Open Source Software: Quality Verification - 9th IFIP WG 2.13 International Conference, OSS 2013, Koper-Capodistria, Slovenia, June 25-28, 2013. Proceedings*, ser. IFIP Advances in Information and Communication Technology, vol. 404. Springer, 2013, pp. 264–270. [Online]. Available: https://doi.org/10.1007/978-3-642-38928-3_20

[23] Eclipse. (2023) Eclipse IDE. [Online]. Available: https://www.eclipse.org/ide/ (Accessed: Feb. 5, 2024).

[24] JSDT. (2023) Javascript Development Tools (JSDT). [Online]. Available: https://www.eclipse.org/webtools/jsdt/ (Accessed: Feb. 5, 2024).

[25] S. O'Hara and R. Slavin, "Modernizing parsing tools: parsing and analysis with object-oriented programming," in *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, N. Grech and T. Lavoie, Eds. ACM, 2019, pp. 20–25. [Online]. Available: https://doi.org/10.1145/3315568.3329967

[26] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2007, Banff, Alberta, Canada, June 25-26, 2007*, J. I. Maletic, A. C. Telea, and A. Marcus, Eds. IEEE Computer Society, 2007, pp. 92–99. [Online]. Available: https://doi.org/10.1109/VISSOF.2007.4290706

[27] D. Moreno-Lumbreras, R. Minelli, A. Villaverde, J. M. González-Barahona, and M. Lanza, "Codecity: A comparison of on-screen and virtual reality," *Inf. Softw. Technol.*, vol. 153, p. 107064, 2023. [Online]. Available: https://doi.org/10.1016/j.infsof.2022.107064

[28] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 551–560. [Online]. Available: https://doi.org/10.1145/1985793.1985868

[29] R. AL-msie'deen, M. Huchard, A.-D. Seriai, C. Urtado, S. Vauttier, and A. Al-Khlifat, "Concept lattices: A representation space to structure software variability," in *2014 5th International Conference on Information and Communication Systems (ICICS)*, 2014, pp. 1–6.

[30] R. A. A. Al-Msie'deen, "Mining feature models from the object-oriented source code of a collection of software product variants," in *Doctoral Symposium of European Conference on Object-Oriented Programming (ECOOP 2013),*, Montpellier, France, July 2013, pp. 1–10.

[31] R. Al-Msie'deen, "Reverse engineering feature models from software variants to build software product lines: REVPLINE approach," Ph.D. dissertation, Montpellier 2 University, France, 2014, [Online]. Available: https://tel.archives-ouvertes.fr/tel-01015102 (Accessed: Feb. 5, 2024).

[32] R. Al-Msie'deen, M. Huchard, and C. Urtado, *Reverse Engineering Feature Models*. LAP LAMBERT Academic Publishing, 2014, iSBN: 9783659614521.

[33] R. Al-Msie'deen, M. Huchard, A. Seriai, C. Urtado, and S. Vauttier, "Automatic documentation of [mined] feature implementations from source code elements and use-case diagrams with the REVPLINE approach," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 24, no. 10, pp. 1413–1438, 2014.

[34] R. A. A. Al-Msie'deen, M. Huchard, A. Seriai, C. Urtado, and S. Vauttier, "Reverse engineering feature models from software configurations using formal concept analysis," in *Proceedings of the Eleventh International Conference on Concept Lattices and Their Applications, Košice, Slovakia, October 7-10, 2014*, ser. CEUR Workshop Proceedings, K. Bertet and S. Rudolph, Eds., vol. 1252. CEUR-WS.org, 2014, pp. 95–106. [Online]. Available: https://ceur-ws.org/Vol-1252/cla2014_submission_13.pdf

[35] R. Al-Msie'deen, A. Seriai, M. Huchard, C. Urtado, and S. Vauttier, "Documenting the mined feature implementations from the object-oriented source code of a collection of software product variants," in *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013*, M. Z. Reformat, Ed. Knowledge Systems Institute Graduate School, 2014, pp. 138–143.

[36] R. Al-Msie'deen, "Tag clouds for object-oriented source code visualization," *Engineering, Technology & Applied Science Research*, vol. 9, no. 3, pp. 4243–4248, 2019. [Online]. Available: https://doi.org/10.48084/etasr.2706

[37] R. A. Al-Msie'deen, "Softcloud: A tool for visualizing software artifacts as tag clouds," *Mutah Lil-Buhuth wad-Dirasat - Natural and Applied Sciences Series*, vol. 37, no. 2, pp. 93–116, 2022.

[38] R. Al-Msie'deen, "Tag clouds for software documents visualization," *International Journal on Informatics Visualization*, vol. 3, no. 4, pp. 361–364, 2019.

[39] R. Al-Msie'deen, H. E. Salman, A. H. Blasi, and M. A. Alsuwaiket, "Naming the identified feature implementation blocks from software source code," *Journal of Communications Software and Systems*, vol. 18, no. 2, pp. 101–110, 2022.

[40] R. Al-Msie'deen, A. H. Blasi, and M. A. Alsuwaiket, "Constructing a software requirements specification and design for electronic IT news magazine system," *International Journal of Advanced and Applied Sciences*, vol. 8, no. 11, pp. 104–118, 2021.

[41] R. A. A. Al-Msie'deen, "A requirement model of local news web/wap application for rural communities," Master's thesis, Universiti Utara Malaysia, Utara, Malaysian, 2008.

[42] R. Al-Msie'deen, "Requirements traceability: Recovering and visualizing traceability links between requirements and source code of object-oriented software systems," *International Journal of Computing and Digital Systems*, vol. 14, no. 1, pp. 1–17, 2023.

[43] H. E. Salman, A. Seriai, C. Dony, and R. Al-Msie'deen, "Recovering traceability links between feature models and source code of product variants," in *Proceedings of the VARiability for You Workshop - Variability Modeling Made Useful for Everyone, VARY '12, Innsbruck, Austria, September 30, 2012*, Ø. Haugen, J. Jézéquel, A. Wasowski, B. Møller-Pedersen, and K. Czarnecki, Eds. ACM, 2012, pp. 21–25. [Online]. Available: https://doi.org/10.1145/2425415.2425420

[44] R. A. Al-Msie'deen and A. H. Blasi, "Software evolution understanding: Automatic extraction of software identifiers map for object-oriented software systems," *Journal of Communications Software and Systems*, vol. 17, no. 1, pp. 20–28, 2021.

[45] R. A. Al-Msie'deen and A. Blasi, "Supporting software documentation with source code summarization," *International Journal of Advanced and Applied Sciences*, vol. 6, no. 1, pp. 59–67, 2019.

[46] R. Al-Msie'deen, *Object-oriented Software Documentation*. Lap Lambert Academic Publishing, 2019, iSBN: 9786200477279.

[47] R. A. Al-Msie'deen, "Automatic labeling of the object-oriented source code: The lotus approach," *Science International-Lahore*, vol. 30, no. 1, pp. 45–48, 2018.

[48] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: A library for implementing analyses and transformations of java source code," *Softw. Pract. Exp.*, vol. 46, no. 9, pp. 1155–1179, 2016. [Online]. Available: https://doi.org/10.1002/spe.2346

[49] M. L. Collard, M. J. Decker, and J. I. Maletic, "SrcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 2013, pp. 516–519. [Online]. Available: https://doi.org/10.1109/ICSM.2013.85

[50] R. Francese, M. Risi, G. Scanniello, and G. Tortora, "Proposing and assessing a software visualization approach based on polymetric views," *J. Vis. Lang. Comput.*, vol. 34-35, pp. 11–24, 2016. [Online]. Available: https://doi.org/10.1016/j.jvlc.2016.05.001

[51] U. Erra and G. Scanniello, "Towards the visualization of software systems as 3d forests: the codetrees environment," in *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, S. Ossowski and P. Lecca, Eds. ACM, 2012, pp. 981–988. [Online]. Available: https://doi.org/10.1145/2245276.2245467

[52] M. Lanza and S. Ducasse, "Polymetric views - A lightweight visual approach to reverse engineering," *IEEE Trans. Software Eng.*, vol. 29, no. 9, pp. 782–795, 2003. [Online]. Available: https://doi.org/10.1109/TSE.2003.1232284

[53] L. Linsbauer, S. Fischer, G. K. Michelon, W. K. G. Assunção, P. Grünbacher, R. E. Lopez-Herrejon, and A. Egyed, "Systematic software reuse with automated extraction and composition for clone-and-own," in *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*, R. E. Lopez-Herrejon, J. Martinez, W. K. G. Assunção, T. Ziadi, M. Acher, and S. R. Vergilio, Eds. Springer International Publishing, 2023, pp. 379–404. [Online]. Available: https://doi.org/10.1007/978-3-031-11686-5_15

[54] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of moose: an agile reengineering environment," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 1–10. [Online]. Available: https://doi.org/10.1145/1081706.1081707

[55] A. Bergel, *Agile Visualization*. LULU Press, 2016.

[56] D. M. Lyons, A. M. Bogar, and D. Baird, "Lightweight multilingual software analysis," in *Proceedings of the 12th International Conference on Software Technologies, ICSOFT 2017, Madrid, Spain, July 24-26, 2017*, J. Cardoso, L. A. Maciaszek, M. van Sinderen, and E. Cabello, Eds. SciTePress, 2017, pp. 201–207. [Online]. Available: https://doi.org/10.5220/0006392502010207

[57] R. Al-Msie'deen, A. Seriai, and M. Huchard, *Reengineering Software Product Variants Into Software Product Line: REVPLINE Approach*. LAP LAMBERT Academic Publishing, 2014.

[58] R. Al-Msie'deen, A. H. Blasi, H. E. Salman, S. S. Alja'afreh, A. Abadleh, M. A. Alsuwaiket, A. Hammouri, A. J. Al_Nawaiseh, W. Tarawneh, and S. A. Al-Showarah, "Detecting commonality and variability in use-case diagram variants," *Journal of Theoretical and Applied Information Technology*, vol. 100, no. 4, pp. 1113–1126, 2022.

[59] D. Kang, T. Kang, and J. Jang, "Papers with code or without code? impact of github repository usability on the diffusion of machine learning research," *Inf. Process. Manag.*, vol. 60, no. 6, p. 103477, 2023. [Online]. Available: https://doi.org/10.1016/j.ipm.2023.103477

[60] L. P. Tizzei, M. O. Dias, C. M. F. Rubira, A. Garcia, and J. Lee, "Components meet aspects: Assessing design stability of a software product line," *Inf. Softw. Technol.*, vol. 53, no. 2, pp. 121–136, 2011. [Online]. Available: https://doi.org/10.1016/j.infsof.2010.08.007

[61] E. Figueiredo. (2011) MobileMedia - java implementation. [Online]. Available: https://homepages.dcc.ufmg.br/~figueiredo/spl/icse08/ (Accessed: Feb. 5, 2024).

[62] R. A. A. Al-Msie'deen. (2023) ScaMaha. [Online]. Available: https://drive.google.com/drive/folders/11_CItJ2pPq_1CAWswcjRsAZnhmQO6OQ8 (Accessed: Feb. 5, 2024).

[63] R. A. Al-Msie'deen, "BushraDBR: An automatic approach to retrieving duplicate bug reports," *International Journal of Computing and Digital Systems*, vol. 15, no. 1, pp. 221–238, 2024.

[64] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *J. Softw. Eng. Res. Dev.*, vol. 5, p. 7, 2017. [Online]. Available: https://doi.org/10.1186/s40411-017-0041-1

[65] Mozilla-rhino. (2023) Rhino: JavaScript in Java - Version 1.7R2. [Online]. Available: https://github.com/mozilla/rhino#rhino-javascript-in-java (Accessed: Feb. 5, 2024).

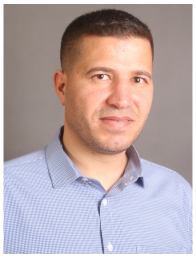
[66] R. A. F. Moreira, W. K. G. Assunção, J. Martinez, and E. Figueiredo, "Open-source software product line extraction processes: the argouml-spl and phaser cases," *Empir. Softw. Eng.*, vol. 27, no. 4, p. 85, 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10104-3

[67] R. Al-Msie'deen. (2018) Drawing shapes software. [Online]. Available: https://sites.google.com/site/ralmsideen/tools (Accessed: Feb. 5, 2024).

[68] Health-Watcher. (2023) Health watcher software - Version 10. [Online]. Available: https://ptolemy.cs.iastate.edu/design-study/ (Accessed: Feb. 5, 2024).

[69] ArgoUML. (2010) ArgoUML software. [Online]. Available: https://argouml-tigris-org.github.io/tigris/argouml/ (Accessed: Feb. 5, 2024).

[70] A. M. Davis, *201 principles of software development*. McGraw-Hill, 1995.



Ra'Fat Al-Msie'Deen is an Associate Professor in the Software Engineering department at Mutah University since 2014. He received his PhD in Software Engineering from the Université de Montpellier, Montpellier - France, in 2014. He received his MSc in Information Technology from the University Utara Malaysia, Kedah - Malaysia, in 2009. He got his BSc in Computer Science from Al-Hussein Bin Talal University, Ma'an - Jordan, in 2007. His research interests include software engineering, requirements engineering, software product line engineering, feature identification, word clouds, and formal concept analysis. Dr. Al-Msie'Deen aimed to utilize his background and skills in the academic and professional fields to enhance students expertise in developing software systems. Contact him at rafatalmsiedeen@mutah.edu.jo. Also, you can reach him using different alternatives: author's page @ github.io, LinkedIn, ResearchGate, or Orcid.