NewbieTOGO: a Light-weighted FAMIX-based Code Visualization Tool for Java Programs

Yuting Bao
UCLA Computer Science
Los Angles, California 90024
ytbao@ucla.edu

Qiujing Lu
UCLA Electrical Engineering
Los Angles, California 90024
qiujing@ucla.edu

Jingyu Shao
UCLA Statistics
Los Angles, California 90024
shaojy11@gmail.com

Xueting Yan
UCLA Computer Science
Los Angles, California 90024
sophiaxueting@gmail.com

ABSTRACT

Maintaining and evolving unfamiliar programs exceeding a certain size are difficult for developers due to lack of comprehensive overview of the entire project structure. This circumstance occurs frequently, especially when newcomers start to work on existing projects. In this report, we present the architecture, implementation and evaluation of our self-developed code visualization tool named NewbieTOGO. It aims at assisting developers to understand the structure of files, classes, methods and attributes, and the inheritance hierarchy among classes. We proposed an algorithm to extract the source code hierarchical information and encode them into JSON file format, and then designed a force-directed graph and a collapsible tree to render the project structure information. We also evaluated our tool by both empirical studies and runtime performance analysis. The results demonstrate that our tool can be helpful for developers to better understand unfamiliar codes and can attract potential users in the future.

CCS CONCEPTS

• **Computer systems organization** → **Software Engineering**; Software Visualization Tools; • **Software** → Code visualization;

KEYWORDS

software visualization, MSE parsing, tree structure representation, inheritance graph

ACM Reference format:
Yuting Bao, Qiujing Lu, Jingyu Shao, and Xueting Yan. 2017. NewbieTOGO: a Light-weighted FAMIX-based Code Visualization Tool for Java Programs. In *Proceedings of Course CS230, LA, CA, USA, June 2017 (Final Project'17),* 11 pages.
https://doi.org/10.475/123_4

Final Project'17, June 2017, LA, CA, USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

1 INTRODUCTION

It is difficult for developers to maintain and adapt software systems when they are extremely large and complex. Due to lack of comprehensive documents from original developers, a tool which can provide a concise and clear overview of the entire structure and relationships between files, classes and methods might be helpful. As the scale of software systems boosts, a proliferation of software visualization approaches in the last decade has been witnessed to support better understanding and further refactoring activities for developers. However, efficient visualization tools and objective assessments of them are still in growing need.

Effective parsing of information extraction from source code is a critical step for code visualization tool. The encoding format should be compact and scalable to adapt to the exponential increment of project size. FAMIX[4] is a common source code meta model family that can represent facts about large scale source codes under analysis. So we design our source code parser based on FAMIX3.0.

Rendering is another essential component in code visualization tools. Visual displays can vividly provide multiple aspects of complicated problems such as high-level structure in parallel, which is an easily acceptable way for human brain. However, a good visualization tool needs to use concise and clear ways to illustrate the most important information without messing up developers. In this report, we proposed an interactive code visualization tool for newcomers to get familiar with existing programs in a short time. Depending on the tree and graph structures, the viewer can extract various kinds of information from our tool NewbieTOGO, i.e., information about the structure of hierarchy dependence, about the size of files, classes, methods and attributes, about the relationships among different modules, about the significance levels of classes, etc. After getting an overview of an unfamiliar project, the viewers can test their understanding by referring to a specific part of source code localized by our tool.

We conducted controlled experiments to test our hypothesis: our visualization tool fits in the gap between newcomers' desire and the services that current visualization tools can provide; our tool offers more readable graph for developers compared to other visualization tools, which contain similar information but represent them in different forms instead of interactive tree structure; our tool provides features that are critical and demanded by developers.

Our major contributions are:

- Designed an algorithm to parse code hierarchy and inheritance relations and encode extracted information into JSON files;
- Rendered the program structure information extracted from source code with a force-directed graph and a collapsible tree;
- Implemented a light-weighted, self-developed code visualization tool based on the algorithm;
- Evaluated our tool by both empirical studies and runtime performance analysis on 7 mature, popular and small-to-large sized JAVA projects.

2 RELATED WORK

2.1 Object-oriented Reverse Engineering

Chikofsky states that "The primary purpose of reverse engineering a software system is to increase the overall comprehensibility of the system for both maintenance and new development "[2]. Indeed, software visualization have long been viewed as a major part to support reverse engineering. Based on this, [7] defines several goals from the view of reverse engineering such as assess the overall quality of the system, gain an overview of the system in terms of size, complexity and structure, locate and understand the most important classes and inheritance hierarchies to get a first impression and a mental model of a system and consider the constraints in the industrial setting, such as simplicity, scalability, language independency for the design their tool.

2.2 Software visualization

Software visualization is defined by Stasko et al. as " The use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software"[11]. It belongs to the filed of Information visualization, which is the study of visual representation of abstract data to reinforce human cognition. Besides viewing code as data, further features in software system are explored. According to Stephan Diehl [3], software visualization can be divided into three parts: Structure, behavior, and evolution. Matrics are designed to show the structure from static code analysis as well as monitored run time behaviors, animated graphs are applied to evolution of whole system over time including changes of modules and developers' contributions. There are also rich forms with which the visualization tools are usually presented with: as plugin for IDE, such as JDepend Maven Plugin, which provides automatically measurement of code quality in terms of extensibility, reusability and maintainability and visualization of matric as user opens a project; as browser-based applications which allows real time analysis and easy deployment and share among collaborators by URL, such as Massey Architecture Explorer2.2.0, an online tool that scales better compared with similar tools; as independent applications that accept intact projects as input and provide stable and detailed analysis on it, such as JArchitect.

2.3 Empirical study of visualization tools

According to [5], there are only empirical studies that show evidence that specific ways of graphical visualization are better than textual visualization for certain tasks. Furthermore, only a few software visualization approaches have been empirically validated so far [6]. One of the reason behind the shortage of empirical evaluation lies in the difficulty of performing a controlled experiment for software visualization, according to [12]. It is almost impossible to reuse the design of other research and difficult to control, common cases are that data may fail to support a true hypothesis or insufficient data lead to statistically insignificant conclusion. Controlled experiment in one method widely applied in the field of evaluation of software visualization. Marcus et al. performed a study to compare 3d visualization tool with sv3d generated from the source code in an IDE and a text file showing metrics values by asking program comprehension questions [9]. Other experiments covered the comparison between enriched and traditional UML views, effectiveness of execution trace visualization tool and evaluation of dynamic object process graphs.

2.4 Tree Structure

There has been extensive work on the exploring effective display of data which have hierarchical properties or attributes. Tree structure is a widely used visual encoding technique that prioritizes readability.Lee [8] shows that it supports exploration of local structure of the graph and expanding the graph as needed. Compared to classic overview techniques which are often limited to revealing clusters, it contains more information and scales better with dense data. There are extensive implementations of tree structure representations such as Gource[1], which is a visualization tool that displays the dirnames, files, users, e.t.c., of a software with an animated tree where root directory is at its center, and directories appear as branches with files as leaves. Another similar tool is Codeswarm[10], which allows the visualization of a long-term tracking of software project progress in nearly real-time with tree structure.

2.5 Other Structures

Treemapping is one of the widely used methods for displaying hierarchy data extracted from code using nested rectangles. However, it will become vague in distinguishing levels when number of classes increase. Heatmap is another frequently used figure presented in the visualization tools, where matrics calculated from code classes are colored in the matrix. Several new structures have been proposed to explore new representation of metrics from code or methods of conveying more information in the same graph. Starting from two dimensional diagrams such as 2D polymetric view, 3D structure tools are developed to visualize code architecture in 3D format, such as CodeCity, which map software matrics into color, size, height of buildings, the whole system is visualized as cities. Codstruction is another 3D software visualization Eclipse plugin tool which is inspired by CodeCity. At the same time, various tools are developed to offer multiple composite views for the same project, such as X-Ray, which is an Eclipse plugin tool that can switch between polymetric views, system complexity view, class dependency view and package dependency views; CodeSonar, which is a well-developed application to present all 3 crucial aspects of a software system: subsystems, interfaces, control flow and potential taint data source.

3 APPROACH

3.1 Overview of Our Approach

To provide an overview of our approach, we first represent the source code based on FAMXI meta-model grammars, and serialize it to generate an *.mse file. We then parse the MSE files through regular expression matching to extract file-to-class and class-to-method/attribute hierarchical information, along with the upper level hierarchy of files and folders acquired by file traversal algorithms, to generate a rich tree structure representation of the whole system. Class dependency relations are also extracted from FAMIX models. We use graph representations to illustrate the inheritance relations between classes and generate the significance level for each node according to their out degrees, which imitates the idea of PageRank. Finally we visualize the acquired information by force-directed graph and collapsible tree.

The overall structure of our system is shown in Figure.1.

3.2 Static Code Analysis

To visualize software systems exceeding a certain critical size, it is important to utilize a scalable and exchangeable format to create a meta-model that contains essential information of the source code. So we use FAMIX models [??] for information extraction in our tool. MSE files have been used to save FAMIX models. MSE is relatively compact and readable, which makes the further parsing of MSE files feasible.

To illustrate MSE format and FAMIX model more clearly, an example MSE file that describes a class in Java is provided as follows. The class named UserTestInput (in file "UserTestInput.java") contains 6 methods and 1 attribute. The entities are sorted in ascending order by their unique identifier. FAMIX.Class is used to define a class entity; FAMIX.FileAnchor can be used to locate the starting and end line of the current type within this class; there also exists a field called "fileName" which assists us in seeking the mapping between source file and class entity.

```
(FAMIX.FileAnchor (id: 171)
    (element (ref: 1412))
    (endLine 36)
    (fileName 'tests/UserInputTest.java')
    (startLine 28))
(FAMIX.Class (id: 760)
    (name 'UserInputTest')
    (container (ref: 410))
    (modifiers 'public')
    (sourceAnchor (ref: 980)))
(FAMIX.FileAnchor (id: 980)
    (element (ref: 760))
    (endLine 91)
    (fileName 'tests/UserInputTest.java')
    (startLine 18))
(FAMIX.FileAnchor (id: 1478)
    ...)
(FAMIX.FileAnchor (id: 1524)
    ...)
(FAMIX.FileAnchor (id: 1885)
    ...)
(FAMIX.FileAnchor (id: 1965)
    ...)
(FAMIX.FileAnchor (id: 2708)
    ...)
(FAMIX.FileAnchor (id: 2953)
    (element (ref: 1135))
    (endLine 20)
    (fileName 'tests/UserInputTest.java')
    (startLine 20))
```

Based on the generated FAMIX meta-model, we generate MSE parsers to extract critical messages for visualization. MSE files contain rich information in aspects of namespaces, packages, classes, methods, and the inheritance relationships, yet lack the hierarchical information between files and folders. So we split the information extraction procedure into two stages: 1) hierarchy between source files, 2) hierarchy of classes and methods /attributes within a single source file. During the second stage, a dependency graph is generated to illustrate the inheritance relationship between classes across files.

Hierarchy Between Source Files. To get the hierarchical status of files and folders in a software system, we traverse through all *.java files inside the root directory by depth first search. The algorithm is illustrated below. Once we reach a leaf node in the search tree, a file parser method is called, which corresponds to the analysis of hierarchical information of classes and methods /attributes within a source file.

```
void DFSWalktree(root, dictionary):
    for file in ListCurrentLevelFiles(root):
        if isDirectory(file):
           subDictionary = {"name":file, "children":[],
            "attribute":value}
            DFSWalktree(file, subDictionary)
          dictionary["children"].append(subDictionary)
            update(dictionary["attribute"])
        elif isRegularFile(file):
            if file.endswith(".java"):
                fileNode = parseFile(file)
                dictionary["children"].add(fileNode)
                update(dictionary["attribute"])
        else:
            Skip
```

Hierarchy Within a Single Source File. We extract file-to-class and class-to-method & attribute mappings from the MSE file to enrich a file leaf node in the previous DFS tree.

To acquire the mappings between a specific file and the classes it contains, we use regular expression match to find all the entities of classes in the MSE file, and extract their anchor files through unique identifier in the entity. For instance, in the MSE file example above, we first match the expression "FAMIX.Class" to localize the class entity, and then find the file "UserTestInput.java" by the reference (*ref: 1*) which points to the anchor file of the class.

Similarly, we extract the class-to-method & attribute relation by the regular-expression-matching rule since the MSE file is well structured. The expression that we utilized for matching is "FAMIX. Method" and "FAMIX.Attribute". To avoid confusion, we add a new

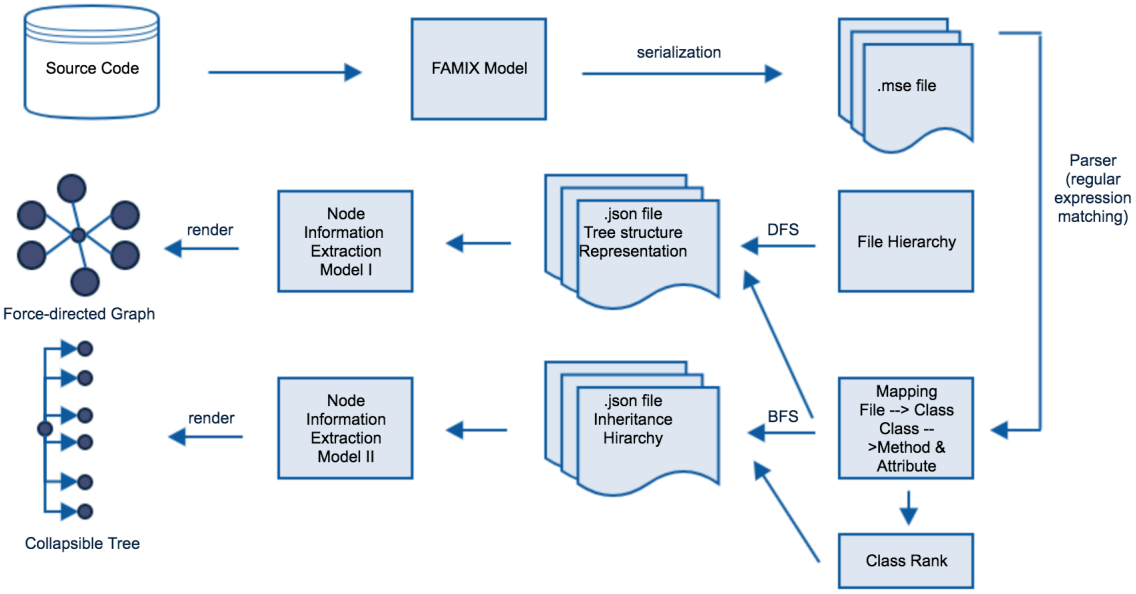


Figure 1: Flow graph of tool NewbieTOGO

field in the node named "Type" to indicate folder, file, class, method and attribute respectively.

Line of Code Extraction. To make full use of the rich information in the MSE file, we also extract lines of code (LOC) and this information can be demonstrated by the radius of nodes in the representation tree. Again we match regular expressions of "startline" and "endline" in the entity that we want to extract the LOC value from, so we get LOC by the simple equation below:

$$LOC = endline - startline + 1 \tag{1}$$

Also we add a new field in each node in the presentation tree and set it radius of nodes accordingly.

Dependency Graph and Class-Rank. So far we have acquired hierarchical information from the physical structure of the whole project, which means that all classes and methods are classified by files and folders. However, when conducting static code analysis, interactions between classes across different files are critical as well. Call relationship between methods depends on runtime analysis and thus is not within our scope.

We extract dependency (inheritance) relationship between classes from the "FAMIX.Inheritance" entity in the MSE file and build a directed dependency graph to illustrate it. In Java programming, although one class can inherit from more than one interface or even combination of one class and several interfaces, multiple inheritance from different classes are not allowed. So we can illustrate the dependent class pairs as superclass (parent) -> subclass(child) in the directed graph, which means subclass is inherited from superclass. Dependency graph is also generated by breadth first search. The algorithm is illustrated as follows.

```
function GenDependGraph(cur_level, EdgeList, dict):
    for cur_element in cur_level:
        if (EdgeList.has_key(cur_element)):
            subdicts = {
            "name": cur_element,
            "children": [], "attribute": Value
            }
            GenDependGraph(
            EdgeList[cur_element],
            EdgeList, ClassDict, subdicts
            )
            dict["children"].append(subdicts)
        else:
            dict["children"].append(
            {"name": cur_element, "attribute": Value}
            )
```

A directed graph is worth of further analysis. Intuitively, a node inherited by more nodes is of more importance. So we can calculate the significance level (score) of each class node through an algorithm similar to PageRank, and then visualize the scores by assigning various colors to the node according to color heatmaps. The PageRank algorithm outputs a probability distribution of stopping at each page when one person randomly picks a page to start and randomly clicks on the links on the pages by simulating random walks. Damping factor is also introduced to add more stability and flexibility into algorithm. 1 minus the value of damping factor can be viewed as the probability of one node jumping randomly on the whole graph instead of visiting neighbors.

$$PR(v_i) = \frac{1-d}{N} + d \sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)},$$

where $v_1, ..v_N$ are the nodes in the graph, $M(v_i)$ is the set of nodes pointing to the node i,$L(v_j)$ is the number of outbound links(edges) on node j. In our case, the procedure of browsing over classes in the dependency graph is similar to the procedure of browsing

pages. In this way, instead of browsing blindly, users can be guided well and directly go to the classes contain top ranks, which they will actually visit most frequently. We first reverse the originally defined dependency graph, pointing the subclass to superclass, then calculate PageRank on this new weighted graph using damping parameter = 0.8. This graph may be disconnected or stuck in certain node, so we set this parameter relative low ,making the simulated random walks' long-distance jumping more radical.

Design of Json files. To better visualize the source code, we generate well-structured *.json files from the information we extracted above as the input of different d3.js models for visualization. JSON is built on a collection of name:value pairs and an ordered list of values. Based on the requirement, we designed an recursive JSON file format to illustrate our tree structure hierarchy of the project. The basic format is as follows:

```
{"name":ModuleName, "attribute":Value, "children":[
  {"name":subModName, "attribute":Value, "children":[
    {"name":subsubModName,"attribute":Value,...},
    {"name":subsubModName,"attribute":Value,...},
    ...]
  },
  {"name":subModName,"attribute":Value,...
  },
  ...]
}
```

Dependency relationship between super and subclasses can also be represented by parent-to-child pointers in JSON file. We can fill in the "attribute" field by any metrics that users are interested in, such as LOC or type of node, which can be easily extracted from FAMIX models.

3.3 Code Visualization

As introduced in the previous section, we use MSE parser to extract two major categories of information from the input software. Given the static code analysis results, we decide to use two customized visualization tools to present different types of information. The first category of information include file hierarchy information and detailed components in each file. The second category of information include conceptual relationship between classes in objected-oriented programming, namely inheritance relationship, and importance score for each file. In the following subsections, we are going to introduce two tools implemented by different d3.js models in order to visualize the code from different perspectives. And we will use our example code "chess" (see appendix) to demonstrate how our tools visualize the code and how to interpret visualized graphs.

Layout Graph. The first part of our tool aims to show the physical layout of a software and detailed components of each file. It provides programmers, who are not familiar with the code, with a high-level structural overview of files relationships and file composition.

The file layout visualization part of our tool has three steps:

(1) Take a JSON from MSE parser as input.
(2) Extract node-level information for each node.
(3) Recursively generate a force-directed graph using D3.js.

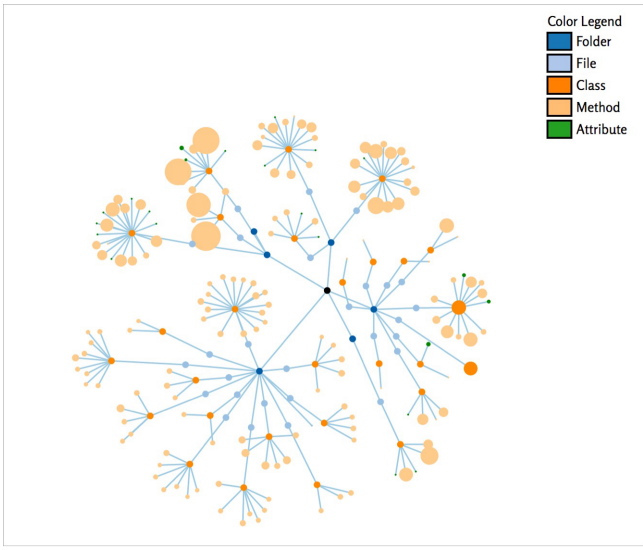


Figure 2: Layout graph that shows file structure layout of sample project chess

Specifically, in the first two steps, our tool would reorder information from the input JSON file in order to prepare node construction in the last step. Here, nodes in the output graph represent different objects, including source code folders, files, Java classes, Java member functions and Java class attributes. Each node contains information, such as node type and size, that is specific to the particular object. In the last step, our tool will use the node-level information to recursively build a force-directed graph using DFS traversal. Node-level information will be used to determine the color and the radius of each node during construction. And our tool will finally outputs a colorful tree as shown in Figure.2.

The central node represents the software folder. Edges from the center node leads to sub-folders and files in the software. A file node can be further expanded to smaller nodes that represents class attributes and member functions. To differentiate various types of nodes in the graph, we color nodes differently. In our design, navy blue nodes represent folders; light blue nodes represent files; orange nodes represent classes; light orange nodes represent class methods; green nodes represent class attributes. Additionally, the size of each node also proportionally represents the number of lines in a file/class/method. In our example code chess, the central root node is connected with five folder nodes: pieces folder, model folder, controller folder, view folder and tests folder. As the software uses MVC model, the tool clearly represents different components in the software. In the test folder, the tool clearly that there are 12 tests files in the tests folder. And further each test file contains a Java class and each class some functions to test different components of the software.

The tool also provides user interactions to allow in-depth study of the code structure q. Initially, the tool expands all nodes, including folder node, file nodes and attribute nodes. If user is interested in a specific node, he can put the mouse on the node, then the node will show a message like "file TextInput.java 198loc". The message

indicates that the node represents a file named TextInput.java and the file contains 198 lines of code. If the user finishes reading detailed information of a class, he can hide the class functions and class attributes into its class node by clicking the class node once. If the user need the class information later, he can click the class node again to expand the node. The collapse and expansion of nodes allow users to control what level of information to show in the graph and keep all useful information only. Furthermore, the tool enables user to drag nodes and place nodes as he desires. This feature provides the user even more freedom in restructuring the code according to the user's understanding.

Inheritance Graph. The goal of implementing the second section of our tool is to visualize the inherited relationships among classes from different files and show the page rank of each class so as to give developer an overall understanding of inherited dependence and importance of classes. The input data is a JSON file containing inherited dependence relationships among classes and page rank scores which is generated from mse parser. In this project, we choose a forest structure to illustrate the inherited relationships among classes. Since in Java, a class can only inherit one class while it can have multiple subclasses. The forest structure is a rational graphical method to present this characteristic. First, we convert the JSON file into node objects and build a collapsible tree structure. To implement the layout of the forest structure, we utilize a JavaScript library named D3.js, which is a powerful tool for producing dynamic interactive data visualizations in web browsers. It combines widely implemented SVG, HTML5 and CSS standards. To show page rank, we decide to utilize the color range to show whether the page rank of a specific class is high or low. With support of D3.js, we implement interactive functionalities like zooming, panning, clicking and dragging in the interface.

Figure 3 shows partial structure of this tool which visualizes the inherited dependence relationship of chess project. In this figure, each node represents a class and each link shows the inherited relationships between two classes (right node inherits from the left node). There are five different colors to represent five score levels of page ranks, i.e., red, yellow, green, sky blue and navy blue, as the importance decreases. From Figure 3, the class "Object" has highest page rank score while classes such as KingTest and KnightTest have lowest page rank scores. Thus, class "Object" is most important class in our example and filled by red while classes like KingTest and KnightTest are corresponding to navy blue nodes. The outline of nodes has two different colors. White outline means all the child nodes of the current node are visible while steer blue outline means the child nodes are hidden. For instance, the nodes JComponent and EventObject have steer blue outlines which means their subclasses are hidden. Click the node by mouse can switch the status between visible and invisible. thus, developers can easily concentrate the parts they care about most and hide other classes. Moreover, drag a node to merge another node can change the inherited relationships.

4 EVALUATION

4.1 Empirical Study

To access the quality of our tool in terms of user experience and efficiency. We compare our tool with some existing visualization tools. The evaluation consists of an empirical study and an efficiency analysis. (The complete survey question is attached in the appendix)

Part I: Evaluation of Overall Performance. We now present the empirical evaluation of our tool NewbieTOGO.Our goal is to conduct surveys to explore the following:

(1) what features are most desired by users when using visualization tools
(2) whether the features are useful for users.
(3) whether the interface of our light-weighted visualization tool is user-friendly

Since users with different background may have diverse evaluation criterion, 2 different questionnaires are designed to collect maximal information from users. We only select the users who use Java as their preferred language. We first explain the functions of our tool and then ask them to use it to visualize an unfamiliar project (spring-core). In the survey, we categorize the users into 2 types: infrequent users, and professional users. Then we hand out questionnaires according to their background.

Table 1: Survey Part I

Questions	Professional users	# Infrequent users
Question 1	4.44	2.92
Question 2	2	6
Question 3	1	3
Question 4	2	8
Question 5	2	6

Table.1 presents conclusive results collected from the survey of 10 participants. There are two columns which stand for professional users (2 participants) and infrequent users (8 participants). We select the following 5 questions of the survey shown in five rows. Q1: What is the average LOC of usual projects (log)? Q2: How many participants have difficulty in understanding overall structure of unfamiliar codes? Q3: How many participants prefer to use visualized method to help understand unfamiliar codes? Q4: How many participants think our tool NewbieTOGO is useful? Q5: How many participants are willing to use our tool NewbieTOGO in the future? From the first row, we can see that participants who often work on large projects have more experiences with code visualization tools. Most of the participants said that they usually have difficulty in understanding overall structure of unfamiliar codes. And about half of them prefer to use visualized method like UML to get an overview of unfamiliar codes. The percentage of number of participants who prefer to use visualized method in infrequent users are slightly lower than that of professional users. The reason can be that some of the infrequent users don't familiar with code visualization techniques. After we shown them our tool Table.1, all of them concede that NewbieTOGO are useful for them to better understand the overall structure and inherited dependence relationships of unfamiliar codes. And most of them are willing to use our tool NewbieTOGO in the future when they need assistance of understanding unfamiliar codes.

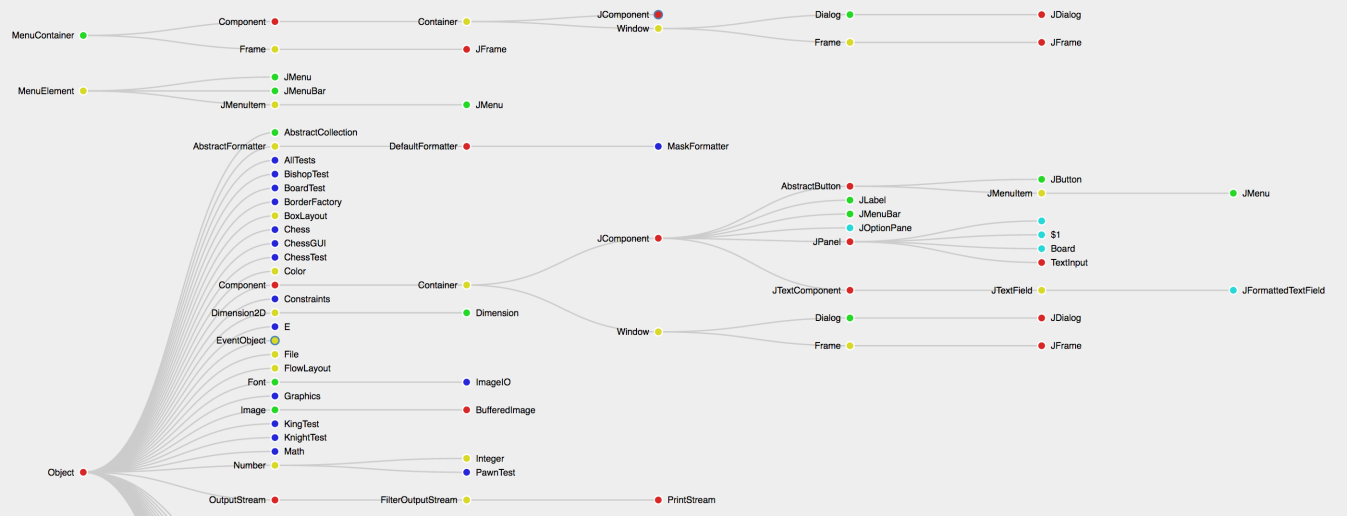
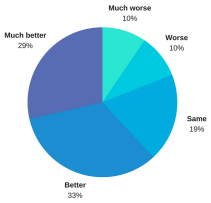
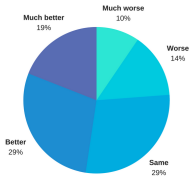


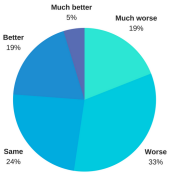
Figure 3: Inheritance Graph Visualization



(a) Inheritance and file layout



(b) Finding important classes



(c) Finding size of files/classes

Figure 4: How is Tool 1's performance compared with Tool 2 in different tasks?

Part II: Comparison of Visual Performance. In this section, we aim to measure the effectiveness of our visualization tool compared with another existing popular visualization tool – Polymetric Views. Polymetric Views provide users with five plots – hotspot diagram, scatter plot, tree map, sunburst diagram and bar chart– given a MSE file as input. In our survey, we asks 21 people in computer science area, give them a new Java software they have never seen before and ask them to use two tools to answer the three categories of questions. 1) What Java classes does class C1 depend on? What classes are there under folder f1? What attributes are there in class C1? 2) What is the largest Java class in terms of logical number of lines in folder f1? 3) Which Java class do you think is the most important one? How confident are you? Is it easy to see? For the convenience of our study, we name our tool as Tool 1 and Polymeric Views as Tool 2. The 12 participants uses different plots and interaction methods in Tool 1 and Tool 2 to understand the Spring source code. After 10 minutes, they fill out our survey on "how much you think Tool 1 outperforms Tool 2 in the above three tasks". Figure 6 shows participants' evaluation of Tool 1 and Tool 2 in the effectiveness in visualizing three tasks respectively.

Figure 6a shows that over majority of participants think Tool 1 is more helpful in answering file layout and class inheritance questions. In the following interview part, majority of think although Sunburst Diagram is able to provide file level structure information, but it does not provide attribute / method level information. Also, Tool 2 do not have any hint in understanding class inheritance relationship. Many participants also complaint the ambiguity of color usage in Tool 2's plots. Figure 6b shows that Tool 1 still outperforms Tool 2 in finding importance classes of a project. Majority of participants believe coloring in Tool 1's inheritance tree helps them to see the importance of each file clearly and is helpful in understanding the core of the project. Figure 6c indicates that although layout graph in Tool 1 uses radius of node to represent the size of a file, Tool 2's bar chart has a clearer representation in checking size. Overall speaking, Tool 1 outperforms Tool 2 in majority of tasks using smaller number of plots with more clear and detailed representations.

4.2 Runtime Performances

In this section, we aim at assessing the runtime performances of our tool on real data of different scales. To gather representative data to assess our tool, we select 7 mature, popular and small-to-large sized Java projects, which are described in Table.2, and use our tool to visualize them.

Table 2: Dataset Description

Dataset	LOC	# Lines in MSE file
Chess	2421	12916
servlet	2569	14859
Android SDK	19630	82226
junit4	46002	203449
cassandra-1.0	181152	698325
cassandra-1.2	246109	954981
cassandra-3.11	NA	1972145

In this performance study, two important aspects are assessed: running time and memory consumption. We use a MacBook Pro Retina with 2.2GHz Intel Core i7 and 16GB of RAM, which is the user-level machine owned by ordinary consumers. We also run another tool *CodeCity*[12] on same datasets for running time comparison.

Running Time Analysis. The 3 main components in total running time of our tool are: 1) Generation of MSE files, 2) Preparations of file2class and class2method mappings, inheritance relation graph, and class ranking scores, 3) Encode hierarchical tree and dependency graph into JSON files by depth and breadth first search. The results are shown in Table.3. MSE parsing time highly relies on the size of MSE files because we need to traverse through all lines to do regular expression matching. Time complexity of DFS and BFS traversal is proportional to the number of nodes in the hierarchy, and thus the time spent on JSON file generation is O(N), where N is the number of leaf nodes(methods and attributes) in our graph representation.

However, the running time of DFS traversal of files in a certain directory actually relies on the depth of the root folder with an exponential time complexity (suppose the branching factor is fixed). When the project size exceeds a certain size, the running time would become extremely large. As illustrated in Figure.5a, the total running time is exploding exponentially with the increment of project size; the correlation between LOC of a certain project and its generated MSE file is linear, as demonstrated by the blue line, which proves that MSE is a compact format.

Table 3: Running Time Analysis (seconds)

Dataset	GenMSE	Parsing	GenJSON	total
Chess	1.912	0.198	0.015	2.125
servlet	2.870	0.266	0.031	3.167
Android SDK	3.302	3.813	0.165	7.28
junit4	4.895	25.560	0.300	30.755
cassandra-1.0	10.334	140.981103	0.57157	157.886
cassandra-1.2	12.975	261.729	0.740	275.444
cassandra-3.11	31.346	> 600	NA	NA

Then we experiment on all the datasets with another tool *CodeCity* and compare their runtime. Because the two tools share the same procedure of generating MSE files (actually we use different FAMIX meta-model versions so our generating time is shorter), we only compare the time spent on MSE parsing and code structure visualization. Table 4 shows the comparison results. Figure 5b illustrates the comparison of running time between *CodeCity* and our tool with the increase of Line of Code (LOC). As we can see from the figure, the processing time is similar when the code size is smaller than 100K. But the trend lines diverge after the project exceeds a certain size, which proves that our tool has lower time complexity and is more light-weighted.

Table 4: Running Time Comparison with *CodeCity* (s)

Dataset	NewbieTOGO(Parsing+JSON)	CodeCity
Chess	0.213	4.8
servlet	0.297	7.8
Android SDK	3.978	12
junit4	25.861	NA
cassandra-1.0	141.552	388.8
cassandra-1.2	262.469	275.444

Memory Usage Analysis. In this part, we use a module named *memory profiler* for monitoring memory usage of a certain function in our tool. Since all the preprocessed data is passed into the hierarchical tree and inheritance graph generation functions as arguments, the inspection of memory usage of DFSWalktree() and GenDependGraph(), which consume the maximum amount of memory during the whole procedure, will be our emphasis.

Table.5 illustrates the space analysis results. *Inc Avg.* represents the average of memory usage differences of the current line with respect to the last one. From Table.5 we can see that, unlike time complexity, space complexity is more stable with respect to project size. Also, there are few increments in memory consumptions between the execution of different line contents.

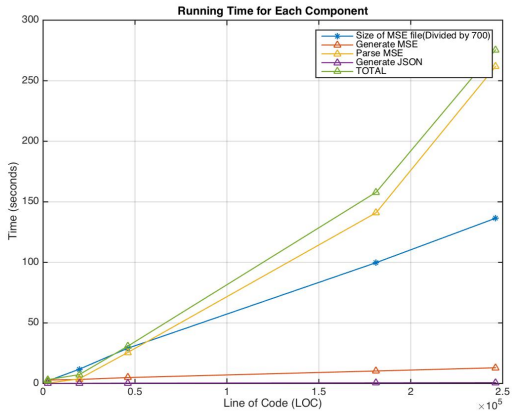
Table 5: Memory Usage Averaged on Each Line (MiB)

Dataset	DFSWalktree	GenDependGraph	Inc Avg.
Chess	58.434	58.461	0
servlet	59.316	59.328	0
Android SDK	68.699	68.711	0
junit4	78.102	78.105	0
cassandra-1.0	96.988	97.000	0
cassandra-1.2	107.652	107.652	0
cassandra-3.11	NA	NA	NA

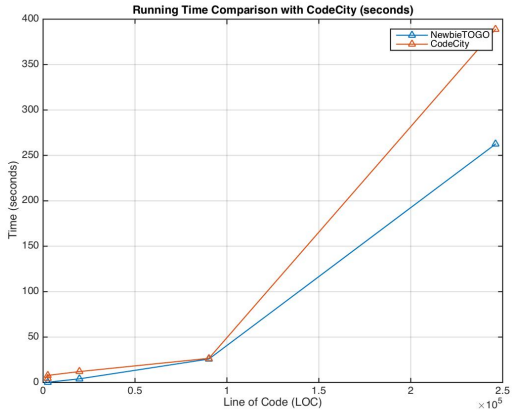
Figure.5c shows the memory consumptions of projects with different size. The space complexity is close to O(log(LOC)) probably due to system memory management optimizations.

5 CONCLUSIONS

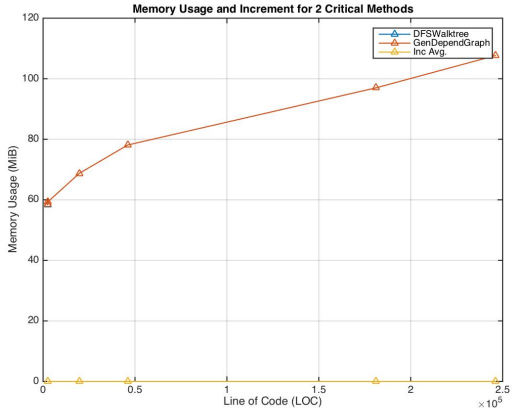
To sum up, our tool NewbieTOGO is a lightweight interactive code visualization tool which can provide a clear and comprehensive overview of code structure to help newcomers understand unfamiliar codes. Our tool is suitable for the following scenario compared to other visualization tools: 1) understand unfamiliar codes, 2) reflect on own projects. Our unique contributions are 1) NewbieTOGO is a lightweight code visualization tool. 2) It does not consider executable files. 3) It provides a vivid and comprehensive interface to



(a) Time (seconds)



(b) Time Comparison(seconds)



(c) Memory (MiB)

Figure 5: Runtime Performance Analysis

display information. 4) It allows viewer to choose a specific part when care about most. 5) We add page rank to analyze significance of classes.

6 DISCUSSION

6.1 Validity Analysis

External Validity. As analyzed above, our tool now only supports software written in Java. The bottleneck lies in our MSE generation program. Currently, our tool can only convert Java source code to MSE files. Therefore, we cannot support other object-oriented programming languages, such as C++, yet.

Internal Validity. In our empirical study, we give participants 10 minutes to answer three categories of questions, which include 10 small questions. It is possible that questions in the survey are biased towards functionalities our tool provides and therefore affect participants' opinions on other compared tools.

Construct Validity. The theory of our evaluation is that if our tool outperforms some popular visualization tools according to some measurements, we can conclude our tool is valuable and can compete against similar visualization tool well. However, it is possible that survey results and runtime analysis in our evaluation are not comprehensive enough to measure the performance of different visualization tools. Therefore, the evaluation cannot definitely demonstrate the superior of our tool against other tools.

Conclusion Validity. We use results from the case study survey to conclude that our tool outperforms the compared tools, such as Polymetric Views. Since the number of participants in our empirical study is not large enough, it is possible that our participants have a biased view and may not give us accurate responses.

6.2 Future Work

Our work is a good starting point in exploring more-detailed, hierarchy representations of Java code. Unlike many other visualization tools that only accept executable as input, we can accept both Java executable and Java source code as input. This feature increases the usefulness of our tool a lot. However, as discussed in the external validity part, we can further find MSE generator for other object-oriented programming languages, such as C++, Python and Ruby. Also, MSE file actually has a lot more other information, we can further improve our MSE parser and visualization interface to add more helpful information to users. In terms of efficiency, the runtime increases faster compared with the increase of project size. Therefore, we can further refactor our MSE generator and parser to make our tool more robust and more scalable.

A APPENDICES

A.1 Evaluation

A.1.1 Empirical Study Survey Questionnaire 1.

Question 1: Have you ever used any visualization tools before?

(Yes) Questionnaire 1:

Part 1: Background Survey

- Which programming language do you prefer?
- Usually how many lines of code are there in your projects? a) <= 500 b) 500 5000 c) 5000 50000 b) >= 50000
- Which visualization tool do you use most frequently?
- Which feature(s) of the visualization tools do you use most frequently?
- What you do usually use visualization tools for?

- Do you think visualization tools help you understand unfamiliar codes faster and better?

Part 2: Concerning Our Tool

- Are there any features that you desire from our tool yet not provided?
- Which feature(s) do you like most about our tool?
- Do you think our tool helps you understand unfamiliar codes faster and better?
- Will you use our tool in the future?

(No) Questionnaire 2:

Part 1: Background Survey

- Which programming language do you prefer?
- Usually how many lines of code are there in your projects? a) <= 500 b) 500 5000 c) 5000 50000 d) >= 50000
- What are the main challenges you come across when reading unfamiliar codes?
- What would you do to understand unfamiliar codes, for example, draw UML?
- Why don't you use visualization tools?

Part 2: Concerning Our Tool

- Are there any features that you desire from our tool yet not provided?
- Which feature(s) do you like most about our tool?
- Do you think our tool helps you understand unfamiliar codes faster and better?
- Will you use our tool in the future?

A.1.2 Empirical Study Survey Questionnaire 2. The survey takes 10 mins. Please use Tool 1 and Tool 2 to find answers to the following sets of questions and compare which tool performs better in different tasks.

Task 1: Finding inheritance and file layout

- What Java class is class AnnotatedComponent extended from?
- How many classes are there in folder spring-framework/spring-core/src/main/java/org/springframework/core/serializer/?
- How many attributes are there in Java class spring-framework/spring-core/src/main/java/org/springframework/core/serializer/DefaultSerializer.java?
- What are the class functions in Java class spring-framework/spring-core/src/main/java/org/springframework/core/serializer/DefaultSerializer.java?

Task 2: Finding important classes

- Which class do you think is the most important one in folder spring-framework/spring-core/src/main/java/org/springframework/core/seriablable?
- What statistics do you use? How confident are you about the previous questions?

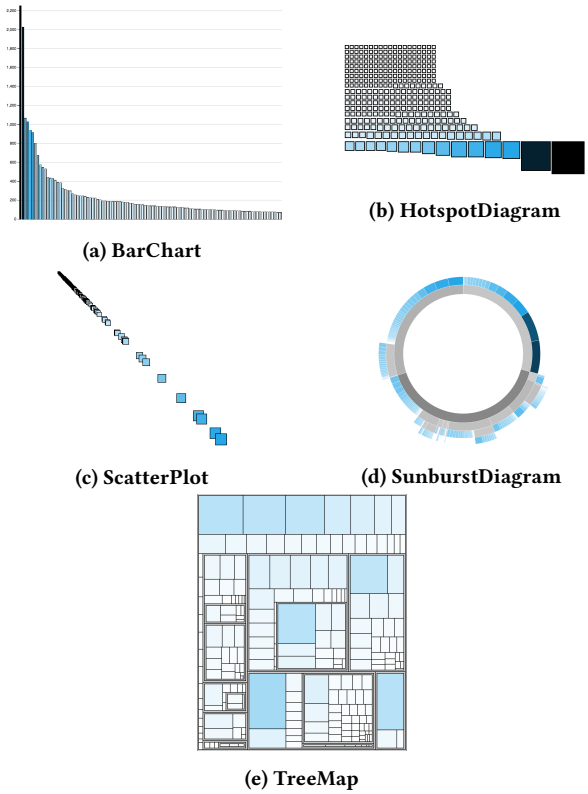
Task 3: Finding size of files/classes

- What is the largest file in folder spring-framework/spring-core/src/main/java/org/springframework/core/seriablable?
- What is the smallest file in folder spring-framework/spring-core/src/main/java/org/springframework/core/env/?

Evaluation:

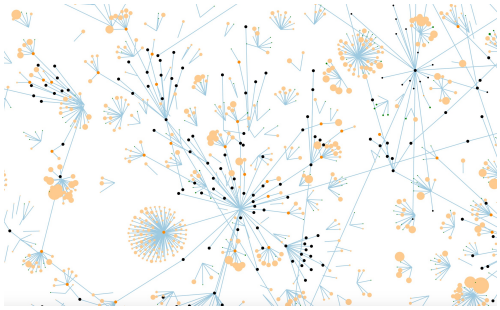
(1) How is Tool 1's performance compared with Tool 2 in Task 1?
   - Tool 1 is much better
   - Tool 1 is better
   - Tool 1 is roughly the same as Tool 2
   - Tool 1 is worse
   - Too1 is much worse
(2) How is Tool 1's performance compared with Tool 2 in Task 2?
   - Tool 1 is much better
   - Tool 1 is better
   - Tool 1 is roughly the same as Tool 2
   - Tool 1 is worse
   - Too1 is much worse
(3) How is Tool 1's performance compared with Tool 2 in Task 3?
   - Tool 1 is much better
   - Tool 1 is better
   - Tool 1 is roughly the same as Tool 2
   - Tool 1 is worse
   - Too1 is much worse

A.1.3 Empirical Study Polymetric View's Plots.

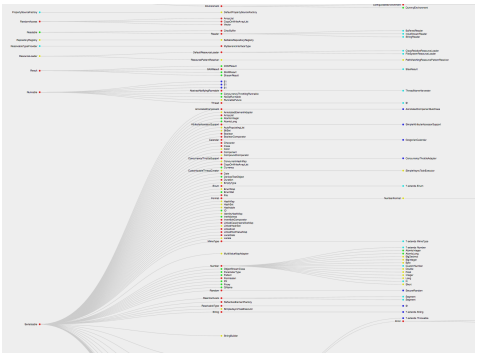


(a) BarChart **(b) HotspotDiagram** **(c) ScatterPlot** **(d) SunburstDiagram** **(e) TreeMap**

Figure 6: How is Tool 1's performance compared with Tool 2 in different tasks?



(a) Layout Graph of spring core



(b) Inheritance Graph of spring core

Figure 7: Visualization results of spring core using NewbieTOGO

ACKNOWLEDGMENTS

The authors would like to thank Prof. Kim and TA Tianyi Zhang for suggestions about our project.

The authors would also like to thank Xinxin Huang for her valuable comments and helpful suggestions.

Individual contributions:

- Yuting Bao: Implemented inheritance graph, conducted empirical study survey part 1, investigated feasible techniques, wrote report.
- Qiujing Lu: Designed empirical survey part 1-2, implemented pagerank algorithm, ran baseline experiments, wrote report.
- Jingyu Shao: Implemented MSE Parser, built tree structure and inheritance graph JSON files, conducted runtime performance analysis, wrote report.
- Xueting Yan: Implemented layout graph, conducted empirical study survey part 2, investigated feasible techniques, wrote report.
- Project url: https://github.com/SophiaYan/CS230-Anonymous

REFERENCES

[1] Andrew Caudwell. 2016. *project web page.* http://gource.io/.
[2] Elliot J. Chikofsky and James H Cross. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE software* 7, 1 (1990), 13–17.
[3] Stephan Diehl. 2007. *Software visualization: visualizing the structure, behaviour, and evolution of software.* Springer Science & Business Media.
[4] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. 2011. MSE and FAMIX 3.0: an interexchange format and source code model family. (2011).
[5] T Dean Hendrix, James H Cross II, Saeed Maghsoodloo, and Matthew L McKinney. 2000. Do visualizations improve program comprehensibility? Experiments with control structure diagrams for Java. In *ACM SIGCSE Bulletin*, Vol. 32. ACM, 382–386.
[6] Rainer Koschke. 2003. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice* 15, 2 (2003), 87–109.
[7] Michele Lanza and Stéphane Ducasse. 2003. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering* 29, 9 (2003), 782–795.
[8] Bongshin Lee, Cynthia Sims Parr, Catherine Plaisant, Benjamin B Bederson, Vladislav Daniel Veksler, Wayne D Gray, and Christopher Kotfila. 2006. Treeplus: Interactive exploration of networks with enhanced tree layouts. *IEEE Transactions on Visualization and Computer Graphics* 12, 6 (2006), 1414–1426.
[9] Andrian Marcus, Denise Comorski, and Andrey Sergeyev. 2005. Supporting the evolution of a software visualization tool through usability studies. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on.* IEEE, 307–316.
[10] Michael Ogawa. 2009. *codeswarm project web page.* http://www.michaelogawa.com/code_swarm/.
[11] John Stasko. 1998. *Software visualization: Programming as a multimedia experience.* MIT press.
[12] Richard Wettel, Michele Lanza, and Romain Robbes. 2011. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 551–560.