University of Stuttgart Visualization Research Center

University of Stuttgart
Allmandring 19
D–70569 Stuttgart

Master's Thesis Nr. 3745

Code Execution Reports: Visually Augmented Summaries of Executed Source Code Fragments

Hafiz Ammar Siddiqui

Course of Study:	Computer Science
Examiner:	Prof. Dr. Daniel Weiskopf
Supervisor:	Dr. Fabian Beck
Commenced:	19 October 2015
Completed:	18 April 2016
CR-Classification:	D.2.3, D.2.5

Abstract

Understanding a fragment of code is important for developers as it enables them to optimize, debug and extend it. Developers adopt different procedures for understanding a piece of code, which involves going through the source code, documentation, and profilers results. Various code comprehension techniques have suggested code summarization approaches, which generates the intended behavior of code in natural language text. In this thesis, we present an approach to summarize the actual behavior of a method during its execution. For this purpose, we create a framework that facilitates the generation of interactive and web-based natural language reports with small embedded word-size visualizations. Then, we develop a tool that profiles a method for runtime behavior, and then it processes the information. The tool uses our framework to generate a visually augmented natural language summary report that explains the behavior of the code. In the end, we conduct a small user study to evaluate the quality of our code execution reports.

Acknowledgement

First of all, I would like to thank God for all the blessing and strength He gave me.

I am very grateful to my parents for their encouragement and support throughout my studies.

I would like to thank my supervisor Dr. Fabian Beck for his continuous supervision, assistance and valuable reviews throughout my work on this thesis.

Contents

List of Figures

List of Tables

1 Introduction

One of the greatest concerns for developers is understanding the behavior of programs. Interpreting the code is necessary for developers as it allows them to optimize or extend a program. Code comprehension also helps in locating the underlying problems and bugs in the code.

There are many techniques that developers employ for understanding the code, which involve various strategies they use before the execution or during the execution of a program. In pre-execution program comprehension, developers go through the source code and the documentation to understand the program [VV95]. For dynamic analysis at runtime, developers use different profiling tools and debuggers to understand the behavior of the program.

1.1 Motivation

Most profiling tools use the program profiling [BL94] approach; they record the behavior of the program during its execution, and then present the recorded data to the developer. Profiling tools generate a lot of data as they cover many details and developers have to go through a lot of information. Going through a large collection of statistical information to establish a meaningful understanding requires some time and effort.

Visualization is an efficient way of communication that uses visual imagery to represent ideas or messages. Information visualization provides a pictorial representation of the data that enable humans to observe patterns and draw useful conclusions from it [YKSJ08]. Quantitative information in graphical form can display more data in little space and can show different levels of detail [TG83].

Humans are comfortable with natural language interface as they communicate with it in their daily life, and it also complements the graphical user interface. It provides effortless and effective communication which results in better understanding [Man98].

Natural language text can be used to describe the behavior of the code to developers. It can summarize the profiling information and can assist developers in understanding the code. Reading the description inside a paragraph is more efficient than scrolling through

tabular data. As it is not possible for natural language text to cover a large amount of data, so software visualization [Die07] techniques can be used to provide visual representations of the numerical data. Multiple representations of information about the runtime behavior of code as a combination of both text and images can provide a better understanding [EP08].

1.2 Goals

Our aim in this thesis is to implement the *Code Execution Reports* tool that describes the runtime behavior of the code as a combination of natural language text and visualizations. Our *Code Execution Reports* tool will summarize the executed behavior of a particular method. More specifically, it will describe the relation of the specified target method with other methods during its execution. It will explain method calls in natural language text. It will embed small word-size visualizations between the text, also known as sparklines [Tuf06]. Word-size visualizations between the words will represent the quantitative information. It will generate a web-based interactive report with embedded word-size visualizations, which will summarize the execution behavior of the target method, and describe its interactions with other methods. The main objectives of thesis are described as follow:

- To create the *Report Building Framework* that generates a web-based interactive report in natural language with word-size visualizations. The framework has to support the basic structure of an English language report and must provide interactive word-size visualizations to depict the quantitative information.
- To create the profiler that records the information of the specified target method during its execution. It has to monitor and record the target method interactions with other methods.
- To create the *Code Execution Reports* tool that records the information related to method calls of a target method using the profiler during its execution, and then uses the *Report Building Framework* to generate a web-based interactive report. The tool has to summarize all the information as a combination of natural language text and word-size visualizations in the report.
- To evaluate the *Code Execution Reports* tool in a small user study and present results.

In this thesis, we have selected the Java programming language not just only for the *Report Building Framework*, but also for the *Code Execution Reports* tool that generates the report with it by profiling execution behavior of a particular Java method.

Figure 1.1 shows an example of the final generated report for the method *drawTreeMap* (see Appendix A for details). The report consists of two section; the first section summarizes the target method behavior overall, and the second section explains the interactions of the target method with other methods in the program during its execution. In the second section, the first paragraph describes the method calls made to the target method, and the second paragraph describes the method calls made by the target method, the last paragraph explains the recursion level depths of the target method.

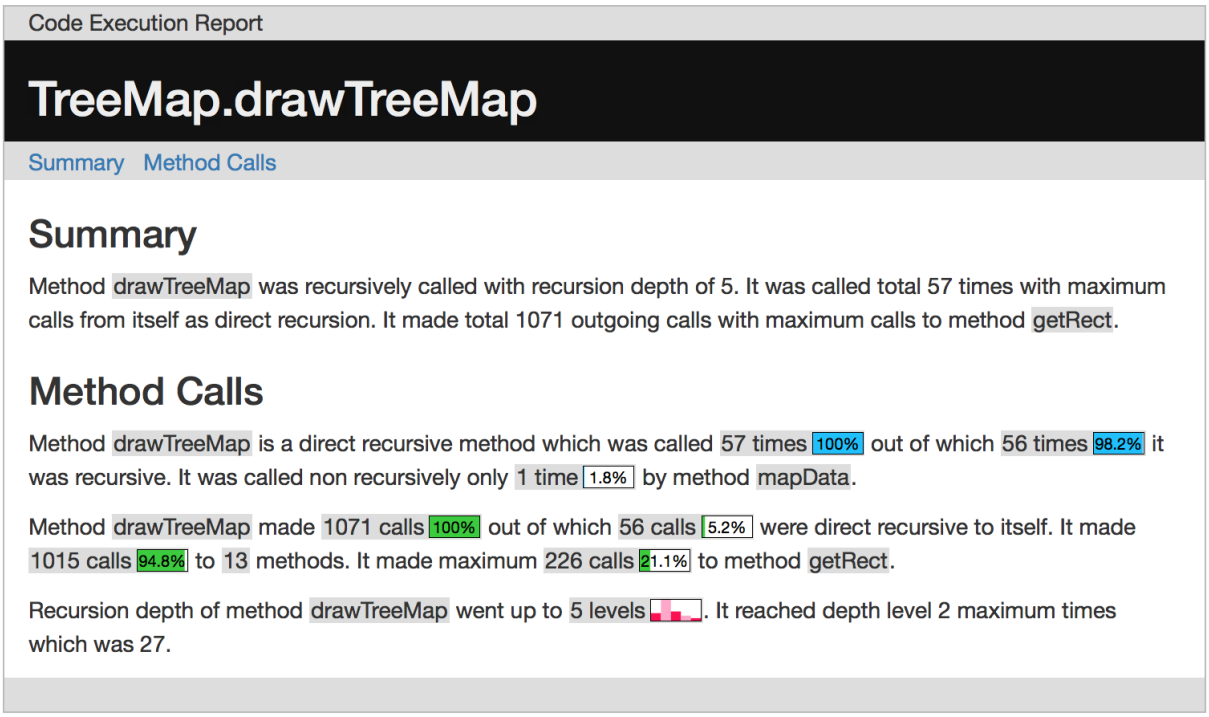


Figure 1.1: Code execution report example

1.3 Organization

The remainder of this thesis consists of the following chapters. Chapter 2 shows the related work in code summarization and visualization. Chapter 3 presents the *Report Building Framework* and explains how it generates the natural language report, and what type of visualizations it supports. Chapter 4 describes the implementation of the *Code Execution Reports* tool and provides insights into profiling and report generation process. Chapter 5 discusses the evaluation results of the tool from the user study. Finally, Chapter 6 draws the conclusion of the presented work.

2 Related Work

Generating summaries in natural language about a fragment of code helps in understanding its functionality. Presently there are no such approaches available that summarize the runtime behavior of a piece of code in natural language, but many code summarization techniques have been proposed to support the code comprehension with the help of natural language summaries by analyzing the source code and other program artifacts.

McBurney and McMillan [MM14] proposed an approach for generating automatic summaries in natural language about the context surrounding the Java methods. The approach identifies the important methods and then extract keywords from those methods. It explains why the method exists.

Moreno *et al.* [MAS+13] presented a technique for creating natural language summaries for Java classes. The technique stereotypes the class, and then generates the human readable description on the base of class stereotype. The technique uses text-based templates to generate the summaries for classes, which is based on Sridhara et al. approach [SHM+10].

Haiduc *et al.* [HAMM10] presented an approach for summarizing the source code using text retrieval techniques. The approach extracts the text from the sources code and converts it into a corpus; then it finds the most relevant terms from the corpus and uses them in the summary.

Rastkar *et al.* [RMM10] worked on the summarization of bug reports and trained the summary generator on bug reports to produce the summaries. Ying and Robillard [YR13] used machine learning approaches for summarizing the code fragments into shorter code fragments.

In the area of software visualization, Harward *et al.* [HIC10] presented simple color coding in the source code editor to deliver information without the overhead. Beck *et al.* [BMDR13] used an approach to augment small word-size visualizations into the code to reflect profiling information. The approach used the static profiling technique to collect the runtime consumption information about the method and method calls. It integrated the visualizations into the source code view, which displays the code and its profiling information in the same view. Baltes *et al.* [BMBD15] extended the approach and added more word-size visualizations in the code editor.

3 Report Building Framework

A software framework is a logical structure that provides a foundation with generic functionality to carry out common and particular tasks. We created the *Report Building Framework* in Java that facilitates the creation of natural language reports in English with word size interactive visual augmentations. The design of framework architecture follows a single pipeline from input data to final report as illustrated in Figure 3.1.

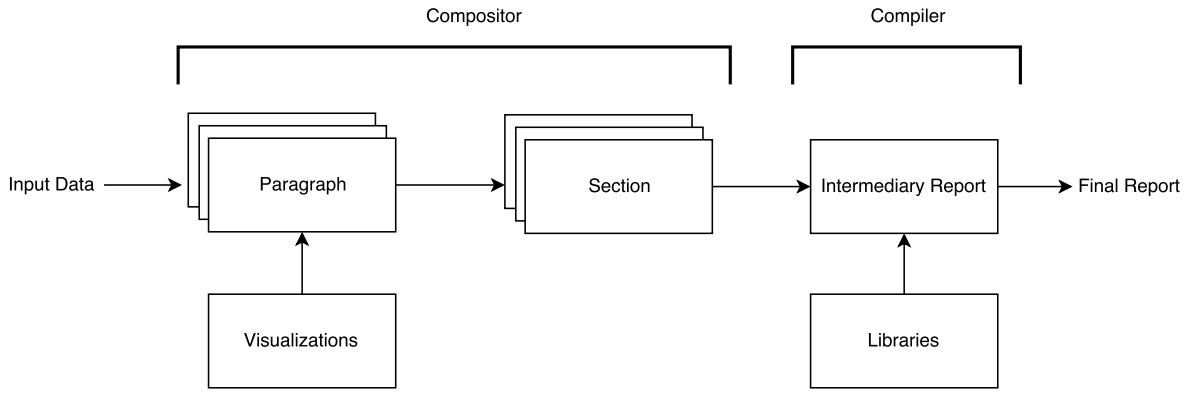


Figure 3.1: Report building framework architecture

The *Report Building Framework* consists of two main components; one is the compositor, and the other is the compiler. It starts with the report compositor, which takes the input data from outside, which could be through a file, service, or by any other means. It creates paragraphs from input data and embeds visualizations inside them in between their words. It composes all paragraphs into sections and results in an intermediary report. The report compiler includes all the necessary libraries and renders the intermediary report into a final web-based report.

3.1 Visualizations

Visualization can represent abstract data through visual imagery. The *Report Building Framework* supports three types of generic, word-size visualizations that developers can insert in between the paragraph words after configuration.

Some visualizations in the framework that represent the information in the pictorial form as vector graphics also contain a small canvas. The height of the canvas is small because it is the same as the font height used in paragraphs of the report, else it will disturb the alignment and appearance of the lines in a paragraph. The width of the canvas is small to maintain an appropriate aspect ratio with respect to its height. The framework provides details-on-demand for visualizations with the help of mouse interactivity. It highlights the area surrounding a visualization, which indicates that further information is available on demand for that particular visualization. Details-on-demand are available as different types of zoom-in levels, which are described as follow:

- **Popover:** A popover is a small overlay that displays more information about a particular visualization in the same view. It uses the mouse over interaction. When a mouse cursor enters the highlighted area surrounding the visualization, it triggers the popover zoom-in level; a popover appears with its content at the top of the visualization if enough space is available, else it is displayed at the bottom. The popover points toward its visualization by using a small arrow. It disappears if mouse cursor leaves the highlighted area.
- **Popup:** A popup is a large window that displays more information about a particular visualization in another view that overlays the previous view. It uses mouse click interaction. When a mouse cursor enters the highlighted area surrounding the visualization, the default mouse cursor changes to a pointing hand shape cursor, which indicates a popup is available for that visualization. Also, the popover that appears here explicitly states "Click for details" in its content. On clicking inside the highlighted area, it triggers the popup zoom-in level; the current view becomes inactive, the popover disappears, and a popup with its content appears in the center of the screen. It disappears if mouse clicks on the cross in the top right corner of the popup or any other area outside the popup.

3.1.1 Progress Bar

A progress bar is a horizontal bar that visualizes the progression of a particular task by filling the bar as it progresses. The filled portion of the progress bar represents the completed work, and the empty part depicts the remaining job. It is a percent-done progress indicator [Mye85], which contains a numerical value to describe the completeness of work using percentage. The *Report Building Framework* provides the word-size progress bar visualization in examples of Figures 3.2 and 3.3.

It includes the following properties, which developers can configure for each progress bar visualization:

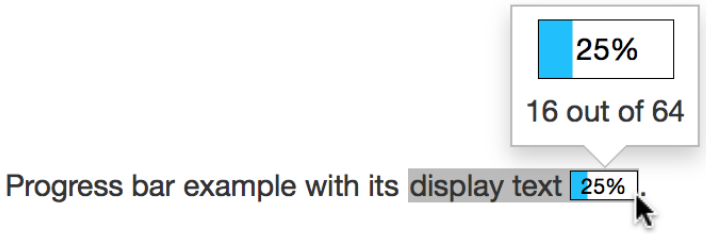


Figure 3.2: Progress bar visualization example with popover

- **Values:** Two numerical values; the first one is the *current value* that defines the amount of completed work, and the second one is the *total value* that specifies the total amount of work. These two values determine the *percentage value*, which appears in the progress bar with the percentage sign.
- **Fill color:** It defines the solid color that paints the progress bar filled portion.
- **Display text:** It appears in front of the progress bar and describes the information in the progress bar visualization.
- **Details-on-demand flag:** If the *details-on-demand flag* is set to true, the framework highlights the surrounding area and also the *display text*. It enables the mouse interaction with the progress bar visualization and shows more information on demand. If the *details-on-demand flag* is false, then it neither highlights the surrounding area nor the *display text*, nothing happens on mouse interactivity and no further information is displayed.

Every progress bar visualization contains a canvas, and the *Report Building Framework* renders that canvas with the progress bar. The framework fills the progress bar by using the *percentage value* as a proportion to calculate the filling portion width out of the total width of the canvas. It inserts the *percentage value* in the center of the progress bar and renders the filled portion with the *fill color* and the empty portion with the white color. It inserts the *display text* before the progress bar. It highlights the progress bar and its *display text* with gray color, only when the *details-on-demand flag* is set to true. Otherwise, it does not highlight. Progress bar visualization supports only one zoom-in level for details-on-demand, and that is popover. On mouse over, the popover shows more information about the progress bar. It enlarges the progress bar graphic inside the popover and also displays the *current value* and *total value* as depicted in Figure 3.2.

Progress bar visualization can show the percentage with decimal point precision up to one digit in a word-size graphic inside a paragraph, and up to two digits in an enlarged graphic inside a popover as shown in Figure 3.3.

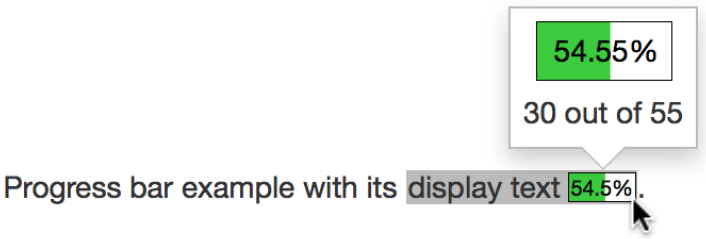


Figure 3.3: Progress bar visualization example with popover on decimal point

3.1.2 Bar Chart

A bar chart is a graph of parallel rectangular bars where each bar represents a category, and its length proportionally depicts the value it holds. A bar chart follows either horizontal or vertical orientation and compares the numerical values of different categories by showing their relative proportions using their lengths. It consists of two axes; the first axis shows the categories, and the second axis shows their values. The *Report Building Framework* provides the word-size vertical bar chart visualization as shown in examples from Figure 3.4 to 3.7.

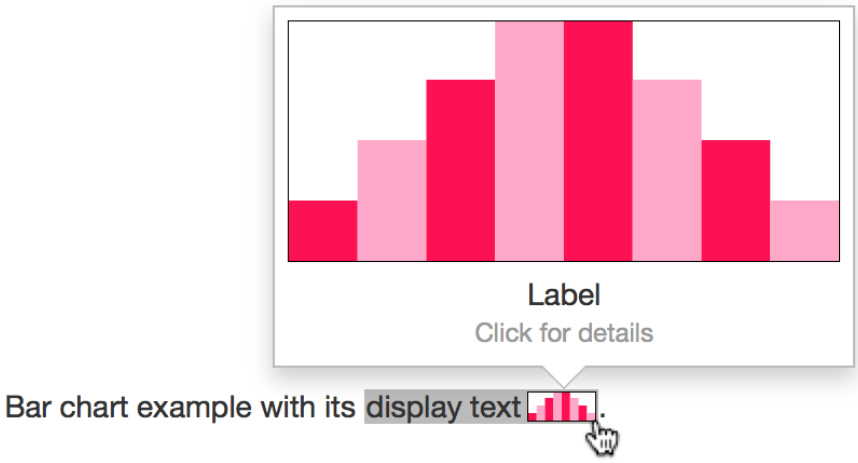


Figure 3.4: Bar chart visualization example with popover

It includes the following properties, which developers can configure for each bar chart visualization:

- **Categories and values:** List of *categories* in which each *category* defines a vertical rectangular bar in the bar chart with its respective numerical *value* that determines the height of that bar.

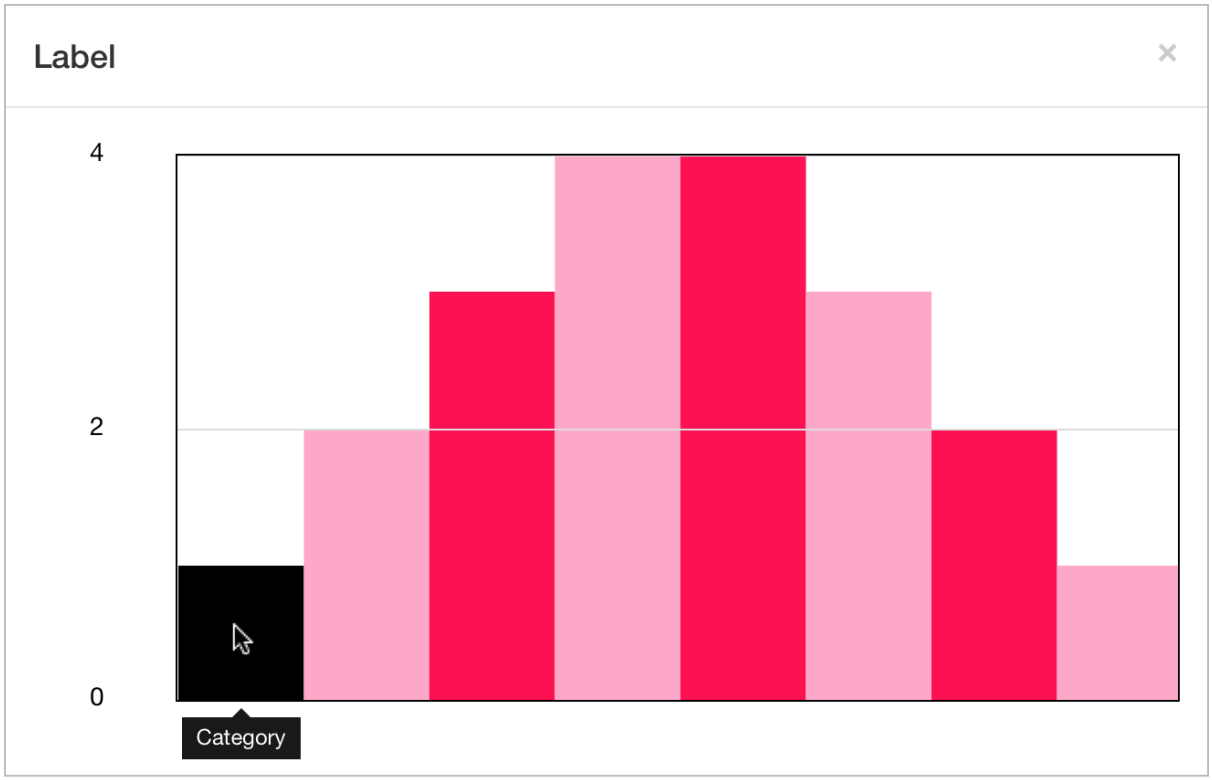


Figure 3.5: Bar chart visualization example with popup

- **Label:** It defines the title of the bar chart visualization.
- **Fill colors:** Two solid colors, one paints the odd bars and other paints the even bars.
- **Display text:** It appears in front of the bar chart and describes the information in the bar chart visualization.
- **Details-on-demand flag:** If the *details-on-demand flag* is set to true, the framework highlights the surrounding area and also the *display text*. It enables the mouse interaction with the bar chart visualization and shows more information on demand. If the *details-on-demand flag* is false, then it neither highlights the surrounding area nor the *display text*, nothing happens on mouse interactivity and no further information is displayed.

Bar chart visualization contains a canvas, and the *Report Building Framework* renders that canvas with the bar chart. The framework generates a vertical bar for each *category* in the bar chart. It evenly splits the width of the canvas among all the bars. It draws each bar with the height proportional to its *value* and scale it down vertically to fit inside

the canvas. It renders the odd and even bars with their respective *fill colors*. It inserts the *display text* before the bar chart. It highlights the bar chart and its *display text* with gray color, only when the *details-on-demand flag* is set to true. Otherwise, it does not highlight. Bar chart visualization supports both zoom-in levels popover and popup for details-on-demand. On mouse over, the popover shows more information about the bar chart. It enlarges the bar chart graphic inside the popover and also displays the *label* and text "Click for details" below the bar chart as illustrated in Figure 3.4. On mouse click, the popup shows the maximized version of the bar chart in another view with the *label* as its title. It generates equally spaced horizontal lines with numerical values as intervals on the left axis and calibrates the height of the bars among them. On further mouse over on any bar inside the popup, a small tooltip appears beneath it, which displays the *category* of that bar as shown in Figure 3.5.

Bar chart visualization can show maximum 50 vertical bars in a word-size graphic and also inside a popover. In the case of more than 50 *categories*, a small right arrow icon appears right after the bar chart in the paragraph. This right arrow also appears below the bar chart inside the popover, which indicates that more *categories* are available in the popup as shown in Figure 3.6. In the popup, two arrows are displayed under the bar chart if *categories* exceed than 50 as shown in Figure 3.7. The two arrows shaped left and right enables horizontal scrolling through all the bars in the bar chart view that appears above them. When the mouse hovers on the right arrow, the bar chart view is scrolled on the right side and shows the bars of the next *categories*. When the mouse hovers on the left arrow, the bar chart view is scrolled on the left side and shows the bars of the previous *categories*. If the mouse cursor is in the bar chart view, then mouse wheel can also scroll the bar chart view, and it also displays the scroll bar while scrolling.

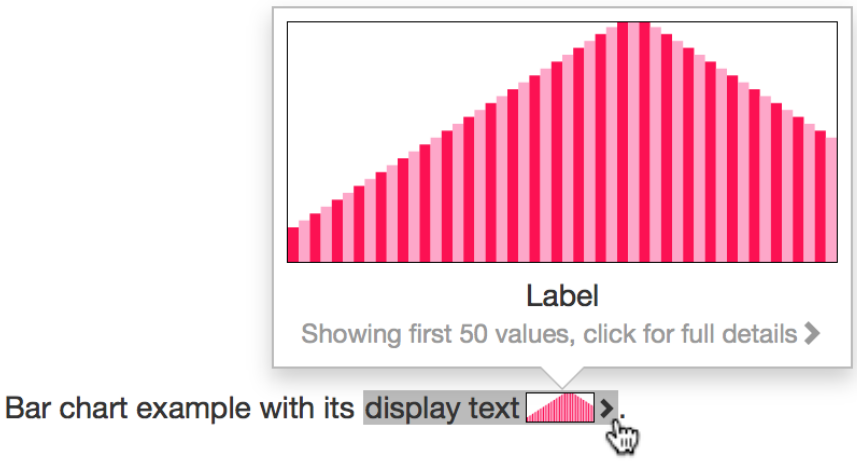


Figure 3.6: Bar chart visualization example with popover on 50+ values

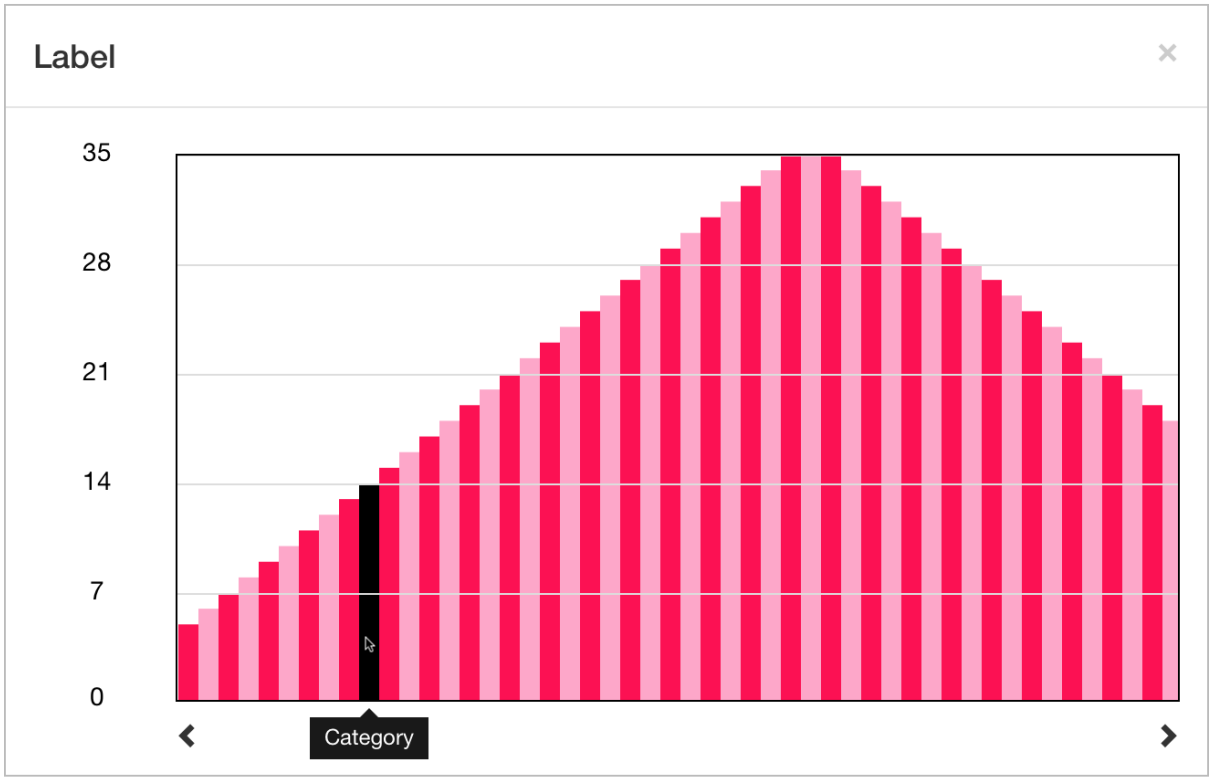


Figure 3.7: Bar chart visualization example with popup on 50+ values

3.1.3 Highlight

Highlighted text shows significance and draws attention towards itself. It emphasizes a particular portion of the text in a sentence or a paragraph and differentiates it from the rest. The *Report Building Framework* provides the highlight visualization for text as shown in examples of Figures 3.8 and 3.9.

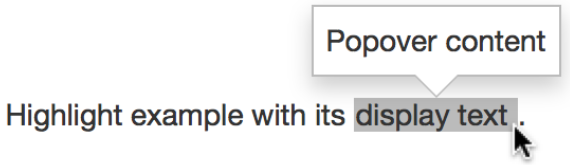


Figure 3.8: Highlight visualization with popover

It includes the following properties, which developers can configure for each highlight visualization:

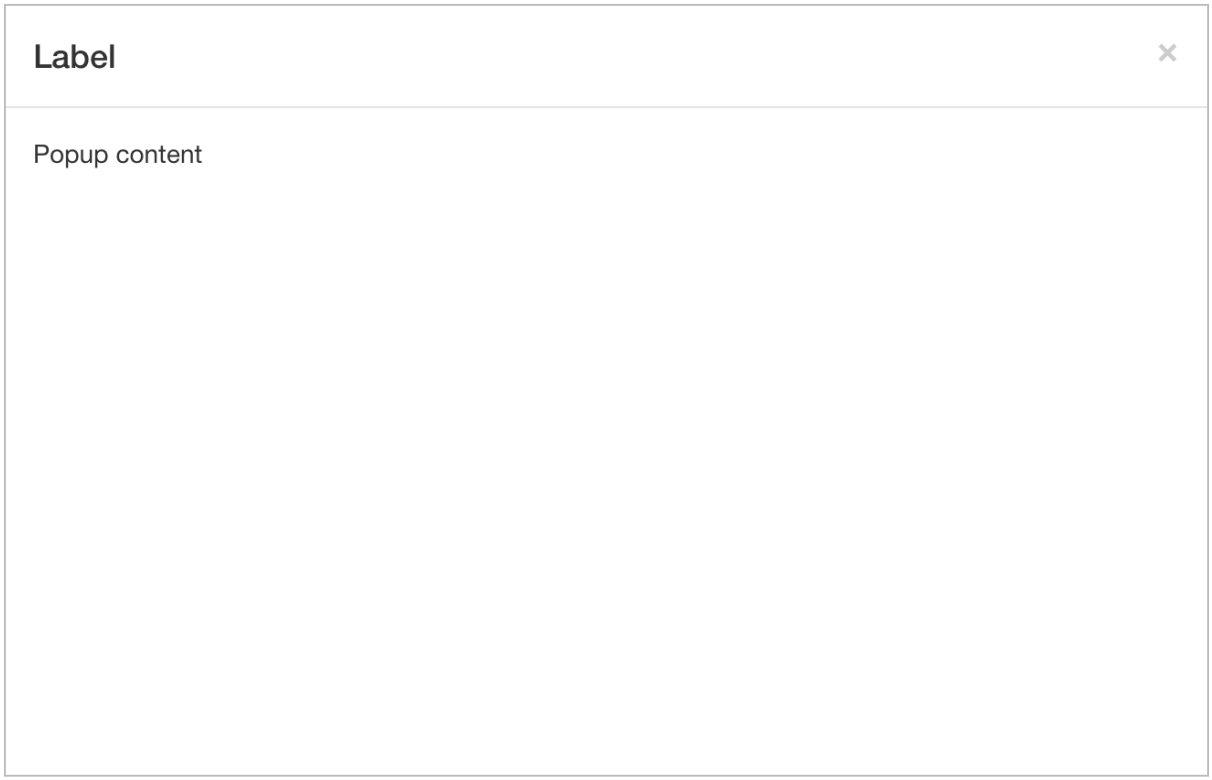


Figure 3.9: Highlight visualization with popup

- **Display text:** It defines the highlighted text that appears in highlight visualization.
- **Label:** It defines the title of the highlight visualization.
- **Zoom-in level:** It can be either popover or popup.
- **Details-on-demand content:** It specifies the content that appears on demand for further information. It can contain text, HTML code and even another nested visualization.

The *Report Building Framework* inserts the *display text* and highlights it by changing its background color to gray. It prepares the *details-on-demand content*; it renders the HTML code in content if there is any, after that it checks for nested visualizations and renders them if it finds any. Highlight visualization supports only one *zoom-on level* as details-on-demand, which can be either popover or popup and are described as follow:

- **Popover zoom-in level:** On mouse over, the popover shows the *details-on-demand content* as shown in Figure 3.8. If there are any nested visualizations in content, then popover only displays them as word-size visualizations without any details-on-demand support.

- **Popup zoom-in level:** On mouse over, the popover shows "Click for details" text and mouse cursor changes to a pointing hand shape cursor, which also indicates a popup is available. On mouse click, the popup shows the *details-on-demand content* in another view with the *label* as its title as illustrated in Figure 3.9. If there are any nested visualizations in content, then popup displays them completely with details-on-demand and mouse interactions support.

3.2 Report Composition

Report composition is the process of putting together all the parts of a report to define the structure and organization of its content. The *Report Building Framework* contains a component by the name of the *report compositor*, which composes the report from sections and paragraphs. The structure of the report is simple as shown in Figure 3.10, it can contain one or more sections, and each section can have one or more paragraphs.

The primary job of the report compositor is to build all the paragraphs in each section, and then define their order and arrangement as per instructions. It outputs an intermediary report after the composition. A paragraph generates text during its build process, and if it does not generate any text, then the report compositor does not include an empty paragraph.

3.2.1 Paragraph

A paragraph is a group of one or more sentences that address a particular idea or topic. It always starts with a new line in a written composition. The *Report Building Framework* enables generating a natural language paragraph using input data at runtime. It further allows embedding a word-size visualization between the words of a paragraph that is also driven by the input data. There is no limit on the length of a paragraph or on the number of visualizations that a paragraph can contain.

Paragraph Graph

A tree data structure represents a group of nodes linked together through directed edges in a hierarchal order that defines the connections between them. Each node in a tree has exactly one parent node, which is present in the hierarchy level above it except the starting root node. A decision tree uses tree-like data structures, and it facilitates the decision making by defining tree nodes as decision points, and the edges to their

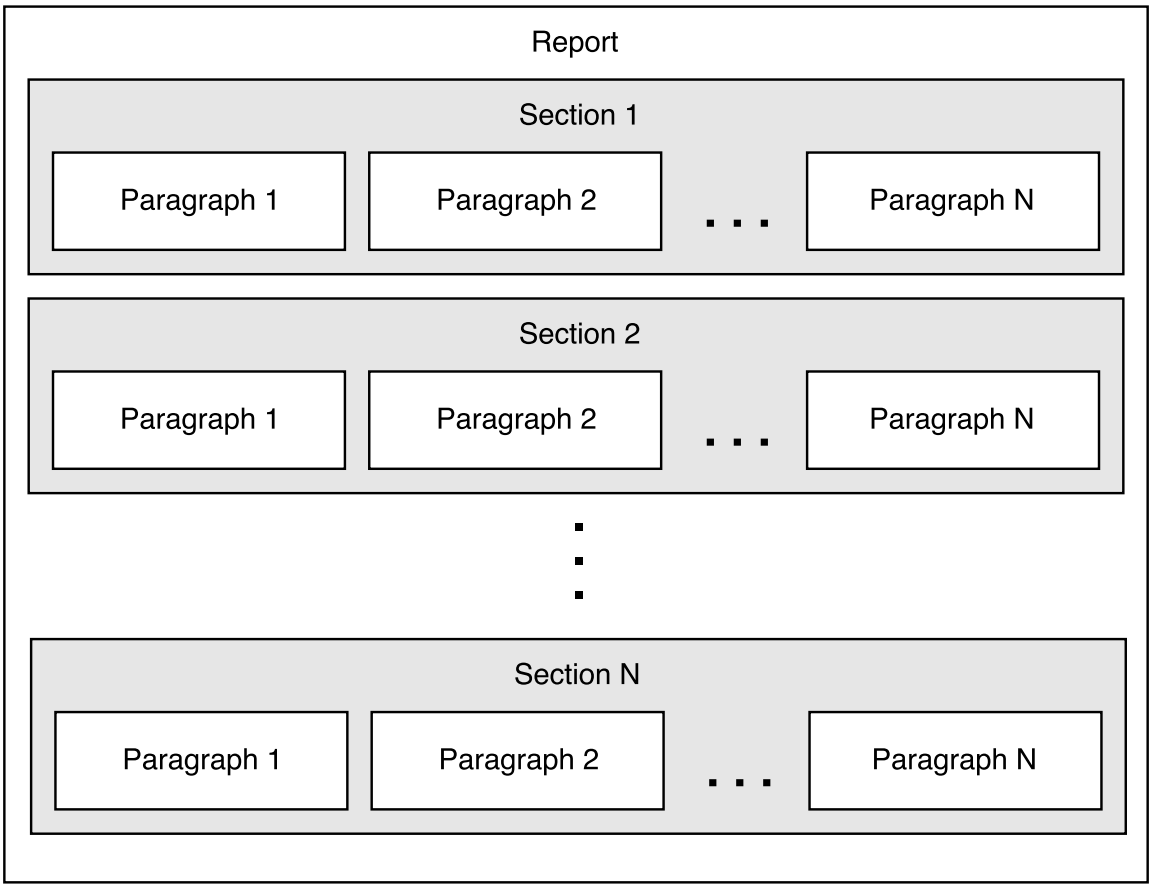


Figure 3.10: Report structure

children as possible outcomes [Oli07]. Using tree data structure in a decision tree for solving complex problems can create unnecessary large trees. Because a node in the tree can not have multiple parents, so decision trees add redundant nodes for two different decision points to diverge to the same outcome.

A directed acyclic graph is similar to a tree, and it allows a node to have multiple parents [BK11]. It can reduce the size of the decision tree due to parent sharing, which can optimize a decision tree. It also reduces the complexity of the decision tree.

For generating natural language text dynamically, the *Report Building Framework* provides a directed acyclic graph for each paragraph, denoted as the *paragraph graph*. A paragraph graph uses a decision tree diagram model [Mor82] for decision making. It consists of two different types of nodes, which are described as follow:

- **Process node:** It contributes the text in a paragraph. It returns a string value, which framework appends in a paragraph. It always contains either a string value

or a method whose return type is string. It has only one child node. Hence, no decision making occurs on its traversal as it has only one child node. A rectangle represents a process node in the diagrams.

- **Decision node:** It does not contribute any text in a paragraph. It has multiple child nodes, and it returns only one child node after decision making. It always contains the conditions for selecting a child node. It only makes the traversal decisions. A rounded rectangle represents a decision node in the diagrams.

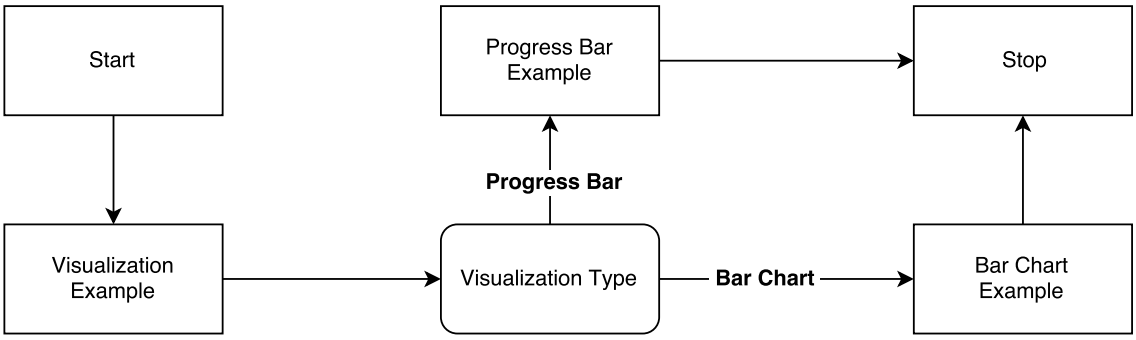


Figure 3.11: Paragraph graph example

A paragraph graph always starts from a start node and ends on a stop node; these two fix nodes are the process nodes that return a null string. Developers create a paragraph graph for every paragraph in a report. They can define the path using decision nodes with their conditions, and the paragraph text using process nodes with their strings. They can use the input data to construct the nodes.

Figure 3.11 shows an example diagram of a paragraph graph, which contains two traversal paths from the start node to the stop node. This paragraph shows the example of word-size visualization in a paragraph. It contains a condition which defines its traversal path and affects its output. The graph traversal starts from the start node which does not contribute any string and only acts as a starting point. The first node it visits is a process node *Visualization Example*; this node adds a string "Visualization example of the" in the paragraph. Next, it visits a condition node *Visualization Type*, now there are two possible outcomes, which depend on the data and the conditions defined by the developers. The possible outcomes of both cases are described as follow:

- **Progress bar:** If the condition evaluates to progress bar, then the next node the graph visits is the *Progress Bar Visualization*. This node adds the progress bar visualization in the paragraph. Next, it visits the stop node which ends the graph traversal. Figure 3.12 shows the paragraph outcome in this case.

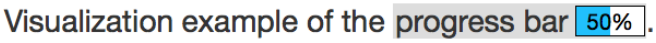


Figure 3.12: Paragraph example with progress bar visualization

- **Bar chart:** If the condition evaluates to bar chart, then the next node the graph visits is the *Bar Chart Visualization*. This node adds the bar chart visualization in the paragraph. Next, it visits the stop node which ends the graph traversal. Figure 3.13 shows the paragraph outcome in this case.

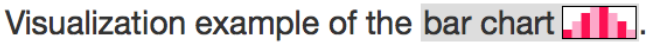


Figure 3.13: Paragraph example with bar chart visualization

Adding a visualization inside a paragraph between words is just like adding another word in a paragraph. Whenever developers create a visualization after configuring its properties, the *Report Building Framework* generates a string representation for that visualization. They can insert that string representation as a word in the process node of a paragraph graph.

Report compositor builds all the paragraphs in the report when it generates the intermediary report. It traverses each paragraph graph from its start node to end node. It produces the paragraph text by going through all the process nodes through a path defined by the conditions nodes. It removes a paragraph from the report if it does not produce any text after traversal. It saves configuration properties and data for each visualization in the same order as they appear in a paragraph.

3.2.2 Section

A section of a report is a group of one or more paragraphs that define a part of that report. The *Report Building Framework* enables creating named sections in a report and then adding paragraphs inside each section. There is no limit on the number of sections in a report or on the number of paragraphs that a section can contain. Each section in the report has a name, which appears at the start of the section in heading above its paragraphs.

Developers specify the order and names of sections and the sequence of paragraphs inside them. When the report compositor generates the intermediary report, it inserts all the paragraphs into their respective sections in their sequence, and then specify the order and name of each section.

3.3 Report Compilation

Report compilation is the process of gathering all the required information and producing the final output. The *Report Building Framework* contains a component by the name of the *report compiler* that renders the intermediary report, and then exports an interactive web-based report. Figure 3.14 shows an example of natural language report generated by the *Report Building Framework* with visual augmentations.

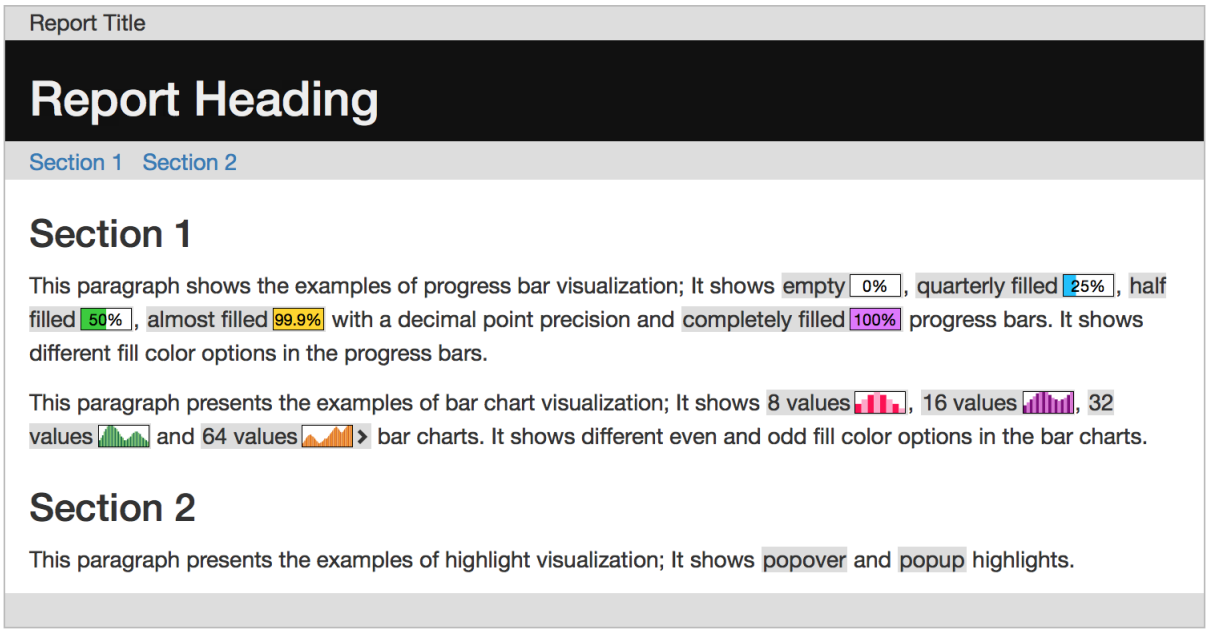


Figure 3.14: Report example

The intermediary report contains the structure and distribution of text with the string representations of visualizations in paragraphs and sections. The primary job of the report compiler is to include all the necessary libraries, and then render the intermediary report content by using those libraries. It uses the following external libraries:

- **Bootstrap:** An open source framework to design the front-end of web applications.
- **jQuery:** An open source library to perform client-side scripting in web applications.
- **D3:** A library to support interactive data visualizations in web applications.

It also includes one internal library of the *Report Building Framework* that overrides some properties of external libraries, and then it uses them to draw interactive visualizations. The report compiler constructs the final report by adding sections with their specified

names in their order. It inserts all paragraphs into their respective sections in their sequence. It goes through each inserted paragraph and gathers their string representations of visualizations. It assigns them unique identifiers, retrieves their saved data and binds it to them. It renders the visualizations with their configuration properties using the library of the *Report Building Framework*, and then it replaces their string representations in paragraphs with them.

It adds report title and report header at the top as defined. It adds a small navigation bar below the header with sections to provide quick navigation to any section. Finally, it exports the complete web-based interactive report at the end.

4 Code Execution Reports

A report is a written document that presents information on any particular topic. We created the *Code Execution Reports* tool in Java that describes the execution behavior of a specific Java method in a natural language report in English. It enriches the report with small word-size interactive embedded visualizations to summarize quantitative information. It explains all the activities and runtime interactions of a particular method with other methods in an interactive visually augmented web-based report. The design of tool architecture follows a single pipeline from Java code to final report as illustrated in Figure 4.1.

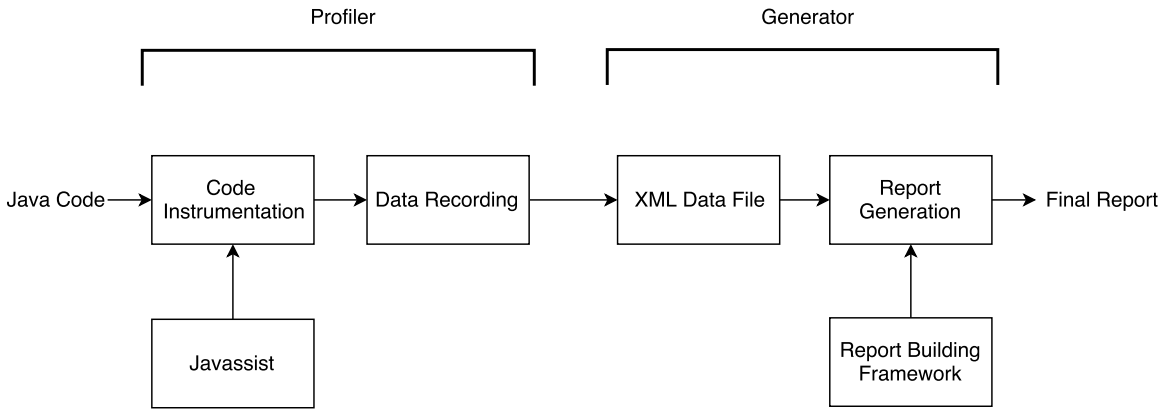


Figure 4.1: Code execution reports architecture

The *Code Execution Reports* tool consists of two main components; one is the profiler, and the other is the generator. It starts with the profiler, which takes the executable Java code with a target method from developers. It instruments the code to gather information about the target method during its runtime. It uses the Javassist [Chi98] library for instrumenting the code. When the instrumented code runs, it records the information about the target method in an XML data file. The generator extracts the profiling information from the data file about the method. It processes the information and uses that as an input data for the *Report Building Framework* (presented in Chapter 3) and generates the final web-based report.

4.1 Method Profiling

Method profiling is the process of examining the behavior of a method during its execution, and then collecting information about it. The *Code Execution Reports* tool contains a component by the name of the *profiler*, which performs method profiling for one specified target method. It monitors and logs the target method interactions with other methods. The profiler uses dynamic byte-code instrumentation [Dmi04] technique to collect the information.

4.1.1 Code Instrumentation

Code Instrumentation is the process of modifying the source code of a program by adding extra lines in it to monitor its dynamic behavior. It is used for debugging, tracing and profiling a program. The profiler in the *Code Execution Reports* tool instruments those parts of the code that are related to the target method. It records the information in a data file during the program execution.

A Java program runs on the Java Virtual Machine [LY99], which is an execution environment that converts the Java bytecode of a program into the machine code. The Java Virtual Machine provides a Tool Interface [OH06], which allows code instrumentation of classes. The Tool Interface takes Java agents in the arguments of the Java Virtual Machine with the program, and then it runs them before the execution of the program. Java agents can manipulate the Java code by using the Java bytecode. The profiler of the *Code Execution Reports* tool is a Java agent that contains classes and methods to instrument, collect, and output the profiling data in a file. It has its executable jar file, which takes the target method in its argument.

As Java agents can instrument the Java code with the help of the Java bytecode so, the profiler component in the tool uses the Javassist library, which simplifies the bytecode manipulation. The Javassist library can transform the Java code into the Java bytecode. It contains data structures to represent the fundamental components of Java programming language like class, method, etc. It facilitates insertion of the custom Java code before any method, after any method, or even at specified line number in a method. It can search for specified expressions in the Java code and can manipulate them.

Developers run the profiler of the *Code Execution Reports* tool with the Java program that contains the method they want to profile for report generation. They pass the profiler as a Java agent in the arguments of Java Virtual Machine and specify the target method in the arguments of the profiler. The Java Virtual Machine runs the profiler before the Java program. The profiler uses an event-driven programming [Fai06] approach to collect the information. It instruments the Java code using the Javassist library and add events

at some specific points in the program, which are method calls to the profiler. It adds the following four type of events:

- **Method start event:** The profiler adds this event at the start of the target method. It passes the depth as an argument to this event, which indicates its depth in the call stack if the target method is in recursion.
- **Method stop event:** The profiler adds this event at all the exit points of the target method.
- **Incoming call event:** The profiler searches for all method calls made to the target method from other methods in the program, and then it add this event right before each method call. It assigns each method that is calling the target method a unique ID, and then it keeps the information of every method with its ID like its name, signature, return type, etc. Finally, it passes the ID in the argument of this event.
- **Outgoing call event:** The profiler searches for all method calls made by the target method to other methods in the program, and then it add this event right before each method call. It assigns each method that the target method is calling a unique ID, and then it keeps the information of every method with its ID like its name, signature, return type, etc. Finally, it passes the ID in the argument of this event.

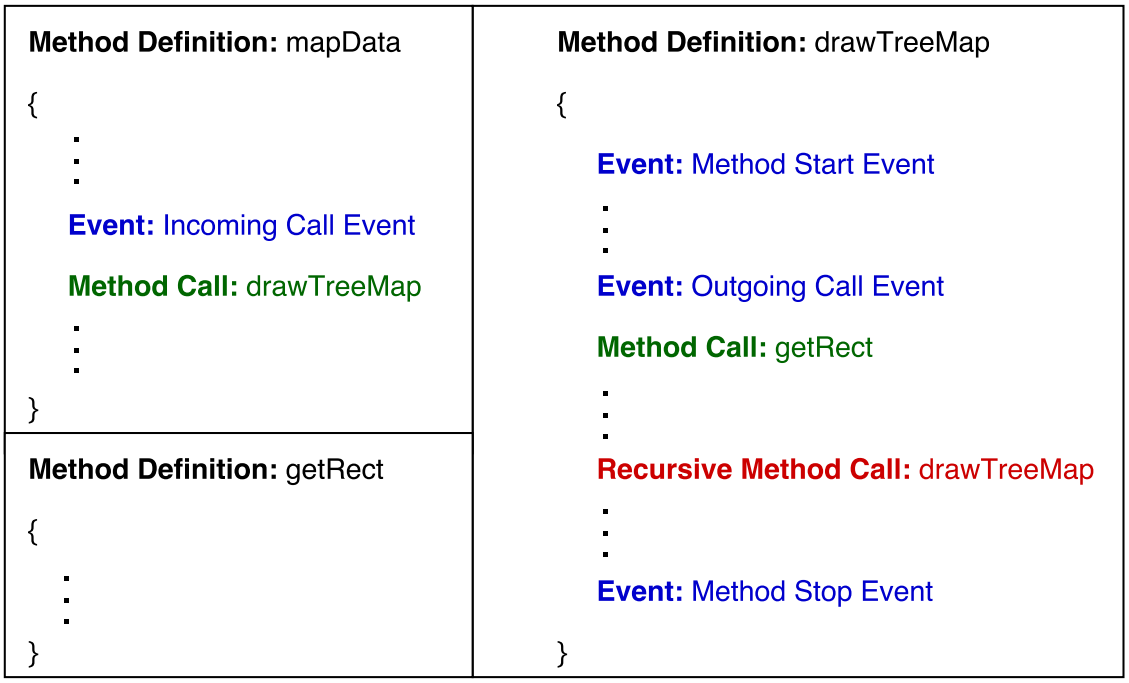


Figure 4.2: Code instrumentation example

Figure 4.2 shows the example of code instrumentation at the abstract level for the method *drawTreeMap* (see Appendix A for details). In this example, method *mapData* calls the target method *drawTreeMap*, which is a recursive method. The target method *drawTreeMap* calls the method *getRect*.

4.1.2 Data Recording

The profiler of the *Code Execution Reports* tool records the runtime behavior of the target method in an XML data file. The Java Virtual Machine first executes the profiler as a Java agent that does the instrumentation of events with the Java code in the program. After that, the Java Virtual Machine executes the program. If the control flow of the program during its execution reaches any event, then it triggers that event, which is a method call to the profiler. The profiler does the event handling during the execution of the program. It writes the data to the file for each *target method execution*. A *target method execution* is explained as follow:

> **Target method execution:** It is the code execution of the target method in the Java Virtual Machine. It starts with the target method's invocation and lasts till the call returns from the target method. The profiler acknowledges one complete *target method execution* from the *method start event* to the *method stop event* in the same thread. Nesting of *target method executions* occurs when the target method is recursive. The profiler records and associates all the runtime activities of the target method with its respective *target method execution*.

Incoming call event occurs when a method calls the target method, which results in an incoming call to the target method. The event provides the ID of this method that calls the target method. The profiler retrieves the information about this method with the help of its ID, and then it updates the information related to this method with details like call frequencies, current thread, etc.

Method start event occurs when the target method starts its execution. The profiler recognizes the start of a new execution of the target method, and it generates a unique ID for this *target method execution*. The profiler maintains a mapping stack for each thread in the program that executes the target method. The profiler pushes this *target method execution* ID to the mapping stack of the current thread. The event provides the depth level information for recursive calls to the profiler as well. The profiler also determines the method which started this *target method execution* with the help of the information it collects with *incoming call events*. It also collects the information when the Java Virtual Machine calls the target method as the entry point of the program directly. If this call of target method is recursive, then profiler goes through the stack trace to

determine the type of recursion. The profiler specifies the recursion type from none, direct recursion, or indirect recursion. It updates the information related to the target method with details like call graphs, call frequencies, recursion, current thread, etc.

Outgoing call event occurs when the target method calls a method, which results in an outgoing call by the target method. The event provides the ID of this method that receives the call from the target method. The profiler retrieves the information about this method with the help of its ID, and then it updates the information for this method with details like call frequencies, current thread, etc. The profiler retrieves the ID of the *target method execution* that called this method with the help of the current thread, and then associate this method with it.

Method stop event occurs when the target method stops its execution in a normal way. The profiler recognizes the execution of the target method is complete. It retrieves the ID of this *target method execution* with the help of the current thread. It retrieves the information about the method which made an incoming call to the target method, and it also obtains the information about the outgoing calls, which target method made to other methods. It writes all information of incoming and outgoing method calls of this *target method execution* into the XML data file.

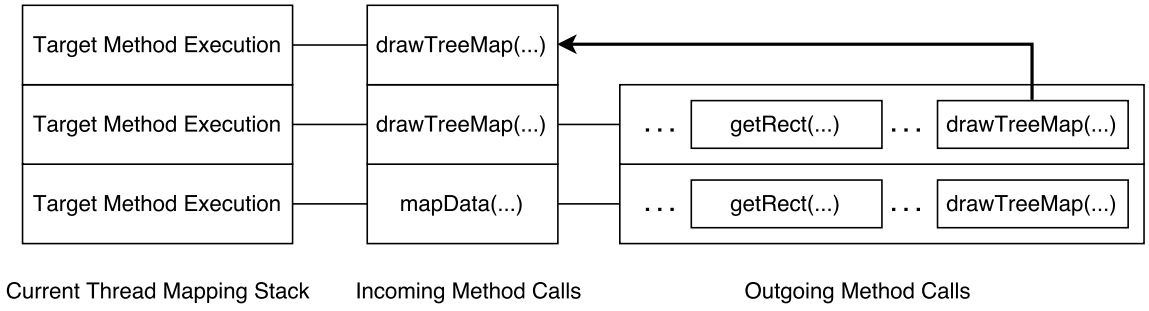


Figure 4.3: Data recording example

Figure 4.3 shows the example of data Recording with a thread mapping stack for the method *drawTreeMap* (see Appendix A for details). In this example, method *mapData* calls the target method *drawTreeMap*, which is a recursive method. The target method *drawTreeMap* calls the method *getRect*.

4.1.3 Profiling Data

The profiler in the *Code Execution Reports* output all the profiling data in an XML file. At the start, the data file contains the list of all the methods that interacts with the target

method in the source code. Then it defines runtime interactions of all the methods with the target method. It also contains the information about call frequencies and call graphs for method calls. The profiler creates the data file as soon as the execution of the program starts and it writes into the file after every *target method execution*.

4.2 Report Generation

Report generation is the process of creating a readable report from the input data in a particular layout and format. The *Code Execution Reports* tool contains a component by the name of the *generator*, which creates the report of a specified method. It describes the execution behavior of a method in natural language with visual augmentations.

The profiler in the *Code Execution Reports* tool first creates the XML data file by profiling a method, which contains the execution behavior of a specified method. Developers provide the data file from the profiler to the generator for creating the final report. The profiler and the generator both use the same schema for XML data file. The generator has its executable jar, which takes the path of the data file as an input in its argument. It parses the data file and retrieves all the information about the execution behavior of the target method. It contains all the necessary data structures to parse and store the information related to Java methods and their interactions. It processes the data and classifies it either as text or visual, and it further categorizes the visuals into appropriate representations. It analyzes the information and draws conclusions from it, which it also presents in the report. The generator uses the *Report Building Framework* to generate the final web-based interactive report in natural language with word-size visualizations.

4.2.1 Data Representation

Data is the collection of statistics, facts, and knowledge. Data representation describes the techniques to display the information. There are various ways to represent the data; it can appear as text, images, audios, videos, or as any combination of these. Data in reports usually appears in the form of text and images, and sometimes also as an aggregation of both. It is not always possible to use only text to summarize the big statistical data. Graphical objects can represent large sets of data in little space. Data visualization of quantitative information is an effective way to communicate; it also depicts the pattern and relationships among the data values [Cle93]. The generator of the *Code Execution Reports* tool uses word-size visualizations from the *Report Building Framework* to summarize the quantitative information. It uses color mapping [Bre94] to distinguish between different categories of data representations in visualizations.

Frequency

Frequency defines the numerical value about the occurrence of a particular event. The profiler in the *Code Execution Reports* tool records various frequencies related to the target method during its execution. The generator component of the *Code Execution Reports* tool categorizes all the frequencies into two major types; one is for incoming method calls, and the other is for outgoing method calls. Both types of frequencies have a total value, which is the sum of all the distributed sub frequencies under them.

- **Incoming method calls frequencies:** These frequencies consists of all those occurrences that are related to method calls made to the target method. The generator divides the all the incoming method calls frequencies into two types. The first type is the recursive method calls frequency that defines all the method calls that the target method got from itself. The second type is the non-recursive method calls frequency that sums up all the method calls made to the target method by other methods. It also includes the frequencies of method calls made by each method individually to the target method. The generator further splits the recursive method calls frequency into two types; one is the direct recursive method calls frequency that defines the invocations of the target method directly by itself, and other is the indirect recursive method calls frequency that sums up the invocations of the target method indirectly via another method. Figure 4.4 shows the incoming method calls frequencies distribution.

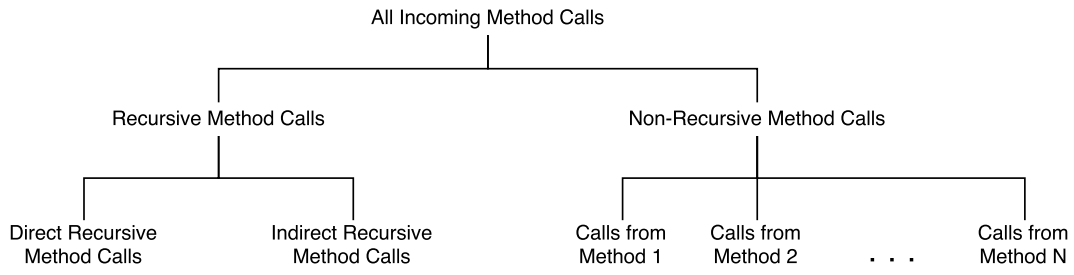


Figure 4.4: Incoming method calls frequencies distribution

- **Outgoing method calls frequencies:** These frequencies consists of all those occurrences that are related to method calls made by the target method. The generator divides the all the outgoing method calls frequencies into two types. The first type is the direct recursive method calls frequency that defines all the method calls that the target method made to itself. The second type is the non-recursive method calls frequency that sums up all the method calls made by the target

method to other methods. It also includes the frequencies of method calls made to each method individually by the target method. Figure 4.5 shows the outgoing method calls frequencies distribution.

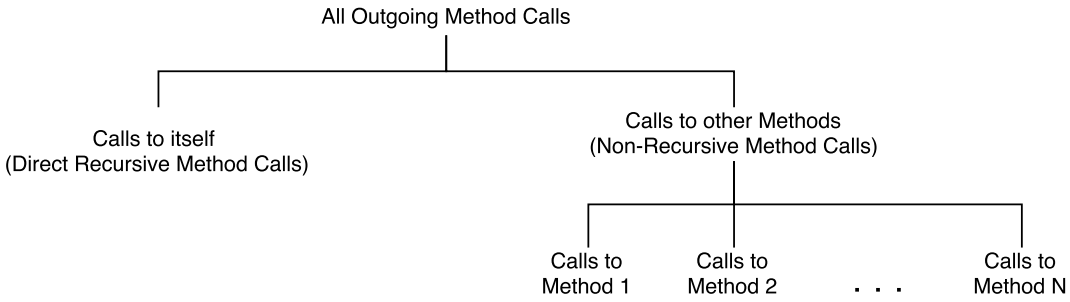


Figure 4.5: Outgoing method calls frequencies distribution

The generator uses the progress bar visualization in the *Report Building Framework* to represent the frequency. It assigns a different color to both method calls frequency categories; a lighter shade of blue color to incoming methods calls frequencies, and a lighter shade of green color to outgoing methods calls frequencies. It uses the color of categories as the *fill color* in their respective progress bar visualization. For every frequency, it uses the value of that frequency as its *current value* and the sum of all frequencies in its category as its *total value*. Hence, the *percentage value* in the progress bar visualization of a frequency represents its share in its category, which gives the idea that how much that frequency contributes overall in its category. The generator puts the actual value of frequency in the *display text* of its progress bar visualization. It uses the word "times" after the frequency value in the *display text* for all incoming method calls frequencies as shown in Figure 4.6. It uses the word "calls" after the frequency value in the *display text* for all outgoing method calls frequencies as shown in Figure 4.7.

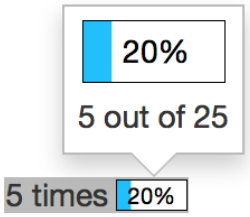


Figure 4.6: Incoming method calls frequency example

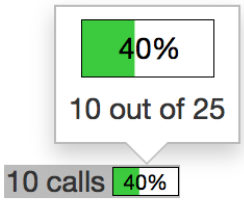


Figure 4.7: Outgoing method calls frequency example

Depth Level

Depth level defines that how deep something goes in nesting. It describes the layer number from the top in the hierarchy. The profiler in the *Code Execution Reports* tool records the depth level of every recursive method call made to the target method during the execution of the program. The generator component of the *Code Execution Reports* counts all recursive method calls made to the target method for each depth level. It determines how many times the target method reached a particular depth level.

The generator uses the bar chart visualization in the *Report Building Framework* to represent the recursion depth levels. It defines a *category* for each depth level in the bar chart visualization. *Categories* start from one and go to the maximum depth level the target method reached during recursion. It assigns the total occurrence count of each depth level as its *value* of its *category* in the bar chart visualization. It assigns different colors to distinguish the representation of recursion depth levels from the frequencies; it defines a lighter shade of red and magenta color as the *fill colors* in the bar chart visualization. The generator puts the maximum recursion depth level reached in the *display text* followed by the word "levels" in the progress bar visualization as shown in Figure 4.8.

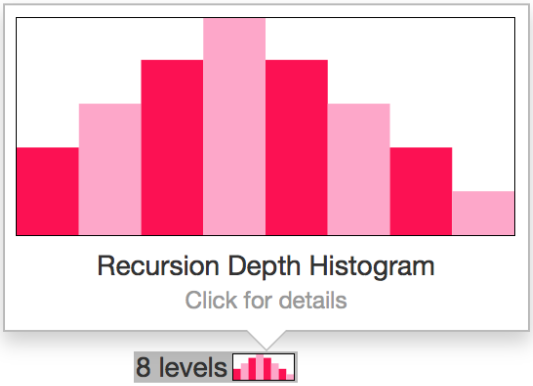


Figure 4.8: Recursion depth levels example

The vertical bar chart from one to maximum recursion depth level with each bar height proportional to the number of times it reached the depth level serves as a histogram. So, the generator uses "Recursion Depth Histogram" as the *label* of the bar chart visualization. It defines each *category* name string with its recursion depth level followed by its count.

Method Name

The method name is an identifier that defines the identity of a method. The profiler in the *Code Execution Reports* tool records the complete details of the target method and all the other methods that interact with it. The details include the method name, class name, package name, parameters, and return type of a method. The generator component of the *Code Execution Reports* tool maintains the complete list of all the methods in the program that interacts with the target method including the target method as well.

Methods in a Java program can have the same name but with different parameters, class, or package. If two methods have the same name in Java, then their parameters must be different. If two methods have the same name and same parameters, then they must have a different class name. If two methods have the same name, same parameters and the same class name, then their classes must belong to different packages.

The generator assigns a display name to each method which represents the method name string uniquely from the rest of the methods in the report. The generator maintains the list of all the methods related to the report, and it ensures the unique string representation of every method name in the list. By default, the display name of every method contains only its name. The generator checks for methods in the list that have the same display name. If it finds any two methods with the same display name, then it starts adding further details in their display names like class name, parameters, return type, package name, etc. It keeps on adding details till both methods display names become unique in the string representation. The generator defines six levels of display names and it starts from level one for each method and goes till its display name becomes unique. The six display levels are described as follow:

1. **Only name:** It only contains the method name.
2. **Short name:** It contains the method name and the class name.
3. **Full short name:** It contains the method name, class name and parameters. It only contains the name of all non-primitive data types in parameters.
4. **Full short name with return:** It contains the method name, class name, parameters, and return type. It only contains the name of all non-primitive data types in parameters. It only contains the name of the return type if its data type is non-primitive.

5. **Full long Name with return:** It contains the method name, class name, package name, parameters, and return type. It only contains the name of all non-primitive data types in parameters. It only contains the name of the return type if its data type is non-primitive.
6. **Full detailed long name with return:** It contains the method name, class name, package name, parameters, and return type. It contains the name, class name and package name for all non-primitive data types in parameters. It contains the name, class name and package name of the return type if its data type is non-primitive.

Figure 4.9 shows an example method with its display names at all six levels.

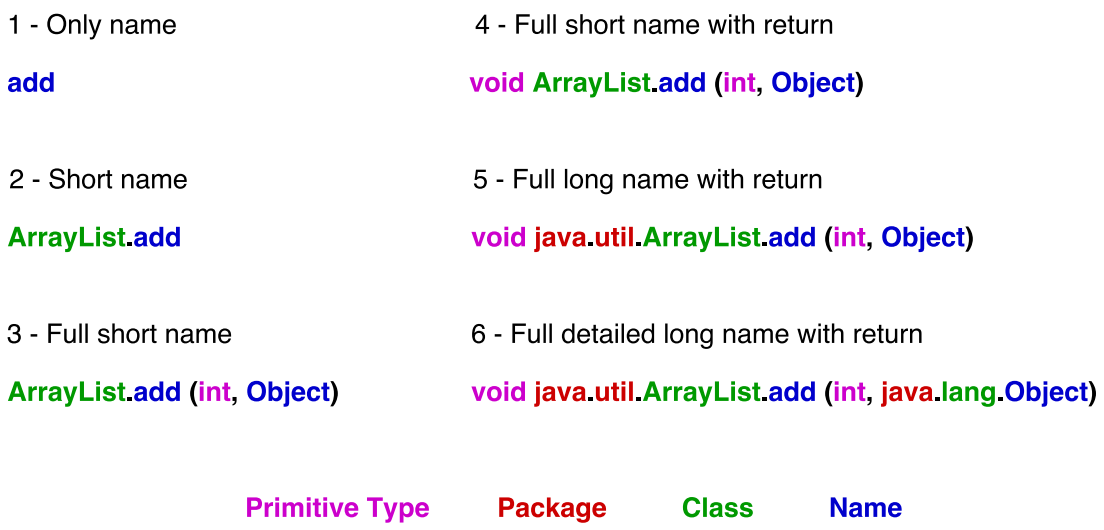


Figure 4.9: Method display names example

The generator uses the highlight visualization in the *Report Building Framework* to represent the method name. It uses the popover *zoom-in level* of the highlight visualization. It is not possible to mention the complete method name with its all details everytime in the report because it will extend the paragraphs of the report unnecessarily long. Also, the report will more look like a Java code rather than a natural language summary. So, the generator only presents the name of the method in the report, which it uses in the *display text* of the highlight visualization. As the report can contain different methods with the same names, and that can cause a problem for the readers if the report only presents the name of the method. So, the generator puts the display name of the method in the *details-on-demand content* in the popover of the highlight visualization as shown in Figure 4.10. If there is no other method with the same name then the generator puts the name and the class name of the method in the popover. But if there is an another

method with the same name, then the generator puts the unique display name in the popover.

Figure 4.10: Method name example

Method Names List

Method names list refers to the representation of more than two methods in a list. The profiler in the *Code Execution Reports* tool records two lists of methods; one is the incoming methods list, and other is the outgoing methods list. The incoming and outgoing methods lists are in the context of the target method. The target method received calls from the methods in the incoming methods list, and the target method made calls to the methods in the outgoing methods list. The generator component of the *Code Execution Reports* tool only mentions the names of maximum two methods from each list. As it summarizes the report in natural language, so it is not possible to mention the names of all the methods in the list. If there are more than two methods in the list, then the generator mentions only one or two significant methods among them and represents the full list of methods using the method names list representation.

The generator uses the highlight visualization in the *Report Building Framework* to represent the method names list. It uses the popup *zoom-in level* of the highlight visualization. It puts the length of the list in the *display text* of the highlight visualization. As *details-on-demand content* can also contain HTML code, so the generator puts an HTML table with two columns in the popup; one represents the method names in the list, and other represents the method call frequencies of methods in the list. It represents the method names using the nested highlight visualization with the popover. As the table does not appear in the report text and it only comes in the popup on demand, so the generator shows more detail about the methods. It starts the display name of the each method from level three in the *display text*, and it puts the unique representation of display name from level four or higher in the popover of the nested highlight visualization. In the case of incoming methods list, it sets the "Method Callers" as the *label* of the highlight visualization and shows the frequencies of incoming method calls. It represents these frequencies with nested visualization for the frequency of incoming method calls category, and it only puts frequency value as its *display text*. In the case of outgoing methods list, it sets the "Called Methods" as the *label* of the

highlight visualization and shows the frequencies of outgoing method calls. It represents these frequencies with nested visualization for the frequency of outgoing method calls category, and it only puts frequency value as its *display text* as shown in Figure 4.11.

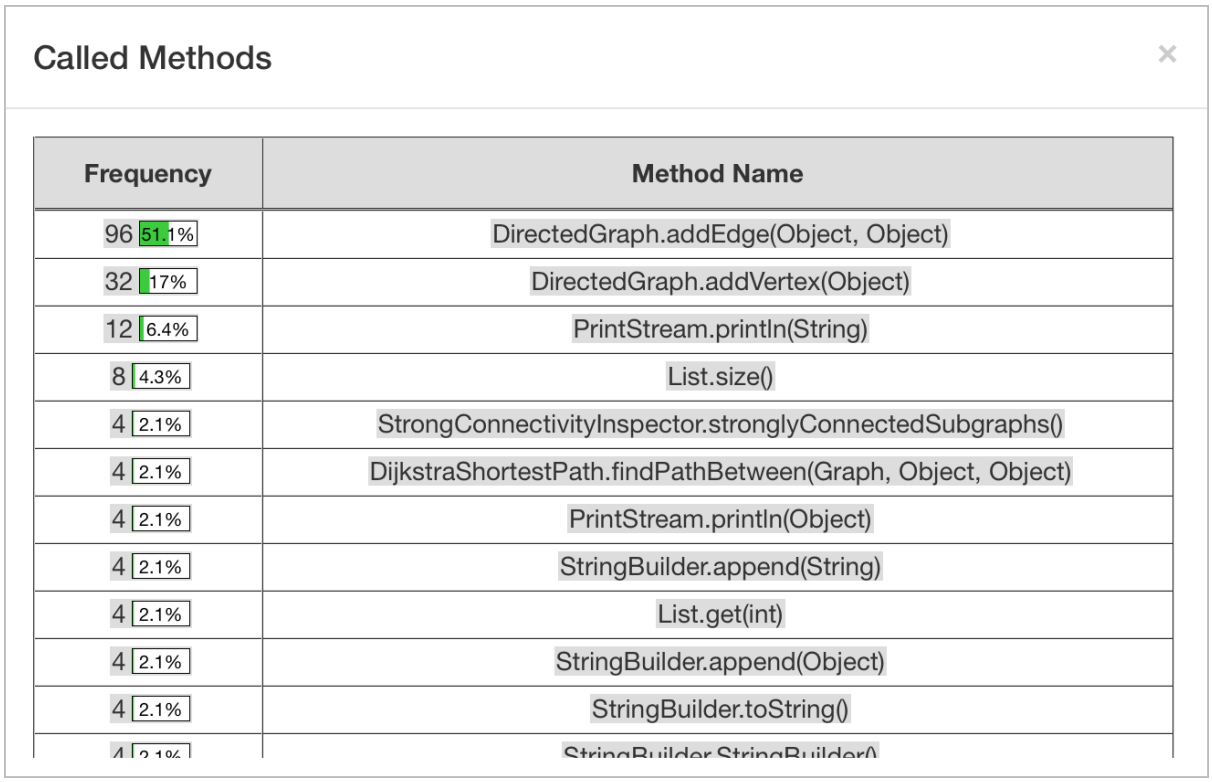


Figure 4.11: Method names list example

4.2.2 Paragaphs

Paragraphs represent all the information in the report. After extracting, processing, and defining the representation of data, the generator component of the *Code Execution Reports* tool summarizes everything in visually augmented natural language text. It combines the text and visual representation of information in paragraphs. It creates paragraph graph of every paragraph in the report for the *Report Building Framework*.

Incoming Method Calls

The incoming method calls paragraph provides the information in natural language text with word-size visualizations about the methods that made calls to the target method. It describes method calls made to the target method. All frequencies in this paragraph

belong to the incoming method calls category. The generator in the *Code Execution Reports* tool summarizes all the information related to the incoming method calls for the target method in this paragraph. This paragraph answers the following questions:

- Did any method called the target method?
- Did the Java Virtual Machine called the target method as an entry point?
- If any of the program methods called the target method, then how many methods, and who are these methods, and how many times they called the target method?
- If there are two or more methods that called the target method, then do they all belong to the same class?
- If all the methods belong to the same class that called the target method, then is it the same class as the target method or different?
- If there are more than two methods that called the target method, then which of these methods called the target method maximum times, and how many times?
- If the target method called itself recursively, then with what type of recursion, and how many times the target method called itself for each type of recursion?
  - Direct recursion
  - Indirect recursion
  - Both direct and indirect recursion
- If target method is a public method, then did any method called the target method from outside the class?

The generator creates the incoming method calls paragraph by using the paragraph graph shown in Figure 4.12 in the *Code Execution Reports* tool. The generator adds the word-size visualizations for method names, incoming method names list, and incoming method call frequencies through the process nodes. The paragraph graph includes the following graph nodes:

- **Start:** It is a process node, but it does not contribute any text in the paragraph. It is the first node in the graph, and the traversal of the graph starts from this node. The next node it returns in the graph is:
  - *Method name*
- **Method name:** It is a process node, and it adds the target method name in the paragraph text. The next node it returns in the graph is:

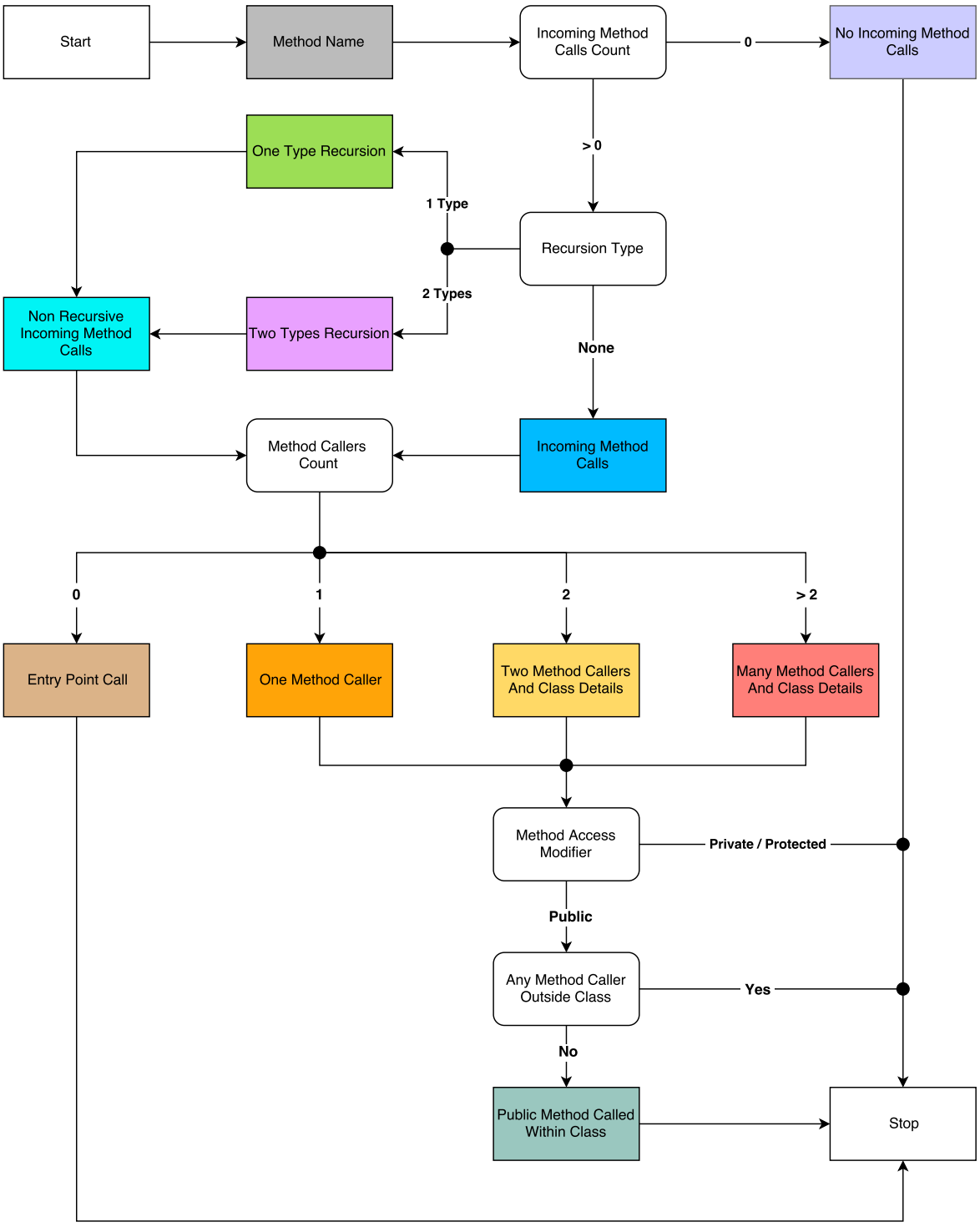


Figure 4.12: Incoming method calls paragraph graph

  - *Incoming method calls count*

- **Incoming method calls count:** It is a decision node. The next node it selects in the graph is:
  - *No incoming method calls:* If no method called the target method.
  - *Recursion type:* If at least one method called the target method.
- **No incoming method calls:** It is a process node, and it describes in the paragraph that no methods called the target method. The next node it returns in the graph is:
  - *Stop*
- **Recursion type:** It is a decision node. The next node it selects in the graph is:
  - *Incoming method calls:* If the target method did not call itself recursively.
  - *One type recursion:* If the target method called itself either direct recursively or indirect recursively, but not both.
  - *Two types recursion:* If the target method called itself both direct recursively and indirect recursively.
- **One type recursion:** It is a process node, and it describes in the paragraph that did the target method called itself direct recursively or indirect recursively, and how many times. The next node it returns in the graph is:
  - *Non-recursive incoming method calls*
- **Two types recursion:** It is a process node, and it describes in the paragraph that how many times the target method called itself direct recursively, and how many times indirect recursively. The next node it returns in the graph is:
  - *Non-recursive incoming method calls*
- **Incoming method calls:** It is a process node, and it describes in the paragraph that how many times any of the program methods called the target method. The next node it returns in the graph is:
  - *Method callers count*
- **Non-recursive incoming method calls:** It is a process node, and it describes in the paragraph that how many times any of the other program methods called the target method non-recursively. The next node it returns in the graph is:
  - *Method callers count*

- **Method callers count:** It is a decision node. The next node it selects in the graph is:
  - *Entry point call:* If the Java Virtual Machine called the target method as an entry point.
  - *One method caller:* If only one method called the target method.
  - *Two method callers and class details:* If two methods called the target method.
  - *Many method callers and class details:* If more than two methods called the target method.
- **Entry point call:** It is a process node, and it describes in the paragraph that Java Virtual Machine called the target method as an entry point. The next node it returns in the graph is:
  - *Stop*
- **One method caller:** It is a process node, and it describes in the paragraph about the only one method that called the target method, and how many times. The next node it returns in the graph is:
  - *Method access modifier*
- **Two method callers and class details:** It is a process node, and it describes in the paragraph about the two methods that called the target method, and how many times, and whether they both belong to the same class, and is it the same class as the target method class or different. The next node it returns in the graph is:
  - *Method access modifier*
- **Many method callers and class details:** It is a process node, and it describes in the paragraph about all methods that called the target method, and how many times, and whether they all belong to the same class, and is it the same class as the target method class or different. It also describes in the paragraph that which of these methods called the target method maximum times. The next node it returns in the graph is:
  - *Method access modifier*
- **Method access modifier:** It is a decision node. The next node it selects in the graph is:

  - *Any method caller outside class:* If the access modifier of the target method is public.
  - *Stop:* If the access modifier of the target method is not public.

- **Any method caller outside class:** It is a decision node. The next node it selects in the graph is:
  - *Public method called within class:* If no method called the target method from outside the class.
  - *Stop:* If at least one method called the target method from outside the class.

- **Public method called within class:** It is a process node, and it describes in the paragraph that the target method is a public method but no method from outside the class called the target method. The next node it returns in the graph is:
  - *Stop:*

- **Stop:** It is a process node, but it does not contribute any text in the paragraph. It is the last node in the graph, and the traversal of the graph ends on this node.

The paragraph graph of incoming method calls generates 651 different templates of natural language text as the paragraph graph tree is traversed from the start node to the stop node. The template does not include the data of method names and method call frequencies; they can be different.

Method saveGraph was called 4 times 100%. It was called 3 times 75% by method updateGraph and only 1 time 25% by method createGraph which belong to the same class as method saveGraph. It is a public method but it was never called from outside its class.

(a) Incoming method calls paragraph

Method saveGraph was called 4 times. It was called 3 times by method updateGraph and only 1 time by method createGraph which belong to the same class as method saveGraph. It is a public method but it was never called from outside its class.

(b) Incoming method calls paragraph with process nodes color coding

Figure 4.13: Incoming method calls paragraph example of a non-recursive method

Figure 4.13 shows an example of incoming method calls paragraph for the method *saveGraph* (see Appendix A for details). Figure 4.13a shows the final output of paragraph with visualizations in the report. Figure 4.13b shows the color-coded text of the paragraph that shows which process nodes contributed what part of the text in the

paragraph. The incoming method calls paragraph graph diagram in Figure 4.12 shows the colors of the process nodes.

Method drawTreeMap is a direct recursive method which was called 57 times 100% out of which 56 times 98.2% it was recursive. It was called non recursively only 1 time 1.8% by method mapData.

(a) Incoming method calls paragraph

Method drawTreeMap is a direct recursive method which was called 57 times out of which 56 times it was recursive. It was called non recursively only 1 time by method mapData.

(b) Incoming method calls paragraph with process nodes color coding

Figure 4.14: Incoming method calls paragraph example of a recursive method

Figure 4.14 shows an example of incoming method calls paragraph for the method *drawTreeMap* (see Appendix A for details). Figure 4.14a shows the final output of paragraph with visualizations in the report. Figure 4.14b shows the color-coded text of the paragraph that shows which process nodes contributed what part of the text in the paragraph. The incoming method calls paragraph graph diagram in Figure 4.12 shows the colors of the process nodes.

Outgoing Method Calls

The outgoing method calls paragraph provides the information in natural language text with word-size visualizations about the method calls that the target method made to other methods. It describes method calls made by the target method. All frequencies in this paragraph belong to the outgoing method calls category. The generator in the *Code Execution Reports* tool summarizes all the information related to the outgoing method calls for the target method in this paragraph. This paragraph answers the following questions:

- Did the target method called any method?
- If the target method called any of the program methods, then how many methods, and who are these methods, and how many times target method called them?
- If the target method called two or more methods, then do they all belong to the same class?
- If the target method called multiple methods and they all belong to the same class, then is it the same class as the target method or different?

- If the target method called more than two methods, then which of these methods the target method called maximum times, and how many times?
- If the target method called itself direct recursively, then how many times?

The generator creates the outgoing method calls paragraph by using the paragraph graph shown in Figure 4.15 in the *Code Execution Reports* tool. The generator adds the word-size visualizations for method names, outgoing method names list, and outgoing method call frequencies through the process nodes. The paragraph graph includes the following graph nodes:

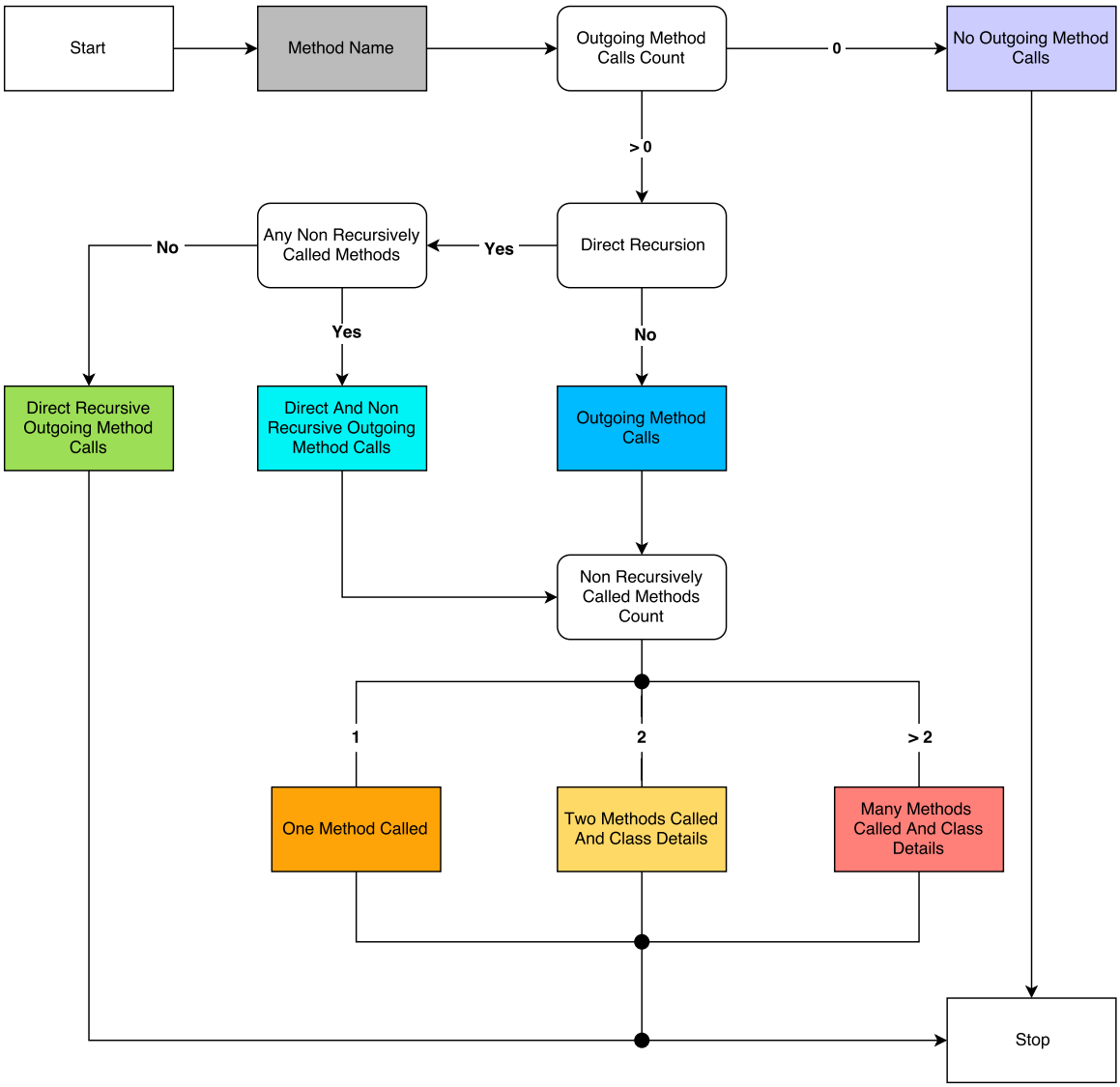


Figure 4.15: Outgoing method calls paragraph graph

- **Start:** It is a process node, but it does not contribute any text in the paragraph. It is the first node in the graph, and the traversal of the graph starts from this node. The next node it returns in the graph is:
  - *Method name*
- **Method name:** It is a process node, and it adds the target method name in the paragraph text. The next node it returns in the graph is:
  - *Outgoing method calls count*
- **Outgoing method calls count:** It is a decision node. The next node it selects in the graph is:
  - *No outgoing method calls:* If the target method did not call any method.
  - *Direct Recursion:* If the target method called at least one method.
- **No outgoing method calls:** It is a process node, and it describes in the paragraph that the target method did not call any method. The next node it returns in the graph is:
  - *Stop*
- **Direct Recursion:** It is a decision node. The next node it selects in the graph is:
  - *Any non-recursively called methods:* If the target method called itself direct recursively.
  - *Outgoing method calls:* If the target method did not call itself direct recursively.
- **Any non-recursively called methods:** It is a decision node. The next node it selects in the graph is:
  - *Direct and non-recursive outgoing method calls:* If the target method called any of the other program methods besides itself.
  - *Direct recursive outgoing method calls:* If the target method did not call any other method besides itself.
- **Direct and non-recursive outgoing method calls:** It is a process node, and it describes in the paragraph that how many times the target method called itself direct recursively, and how many times it called any of the other program methods. The next node it returns in the graph is:
  - *Non-recursively called methods count*

- **Outgoing method calls:** It is a process node, and it describes in the paragraph that how many times the target method called any of the program methods.
  - *Non-recursively called methods count*
- **Non-recursively called methods count:** It is a decision node. The next node it selects in the graph is:
  - *One method called:* If the target method called only one method other than itself.
  - *Two methods called and class details:* If the target method called two methods other than itself.
  - *Many methods called and class details:* If the target method called more than two methods other than itself.
- **One method called:** It is a process node, and it describes in the paragraph that the target method called only one method, and how many times. The next node it returns in the graph is:
  - *Stop*
- **Two methods called and class details:** It is a process node, and it describes in the paragraph that the target method called two methods, and how many times, and whether they both belong to the same class, and it is the same class as the target method class or different. The next node it returns in the graph is:
  - *Stop*
- **Many methods called and class details:** It is a process node, and it describes in the paragraph that the target method called multiple methods, and how many times, and whether they all belong to the same class, and it is the same class as the target method class or different. It also describes in the paragraph that which of these methods the target method called maximum times. The next node it returns in the graph is:
  - *Stop*
- **Direct recursive outgoing method calls:** It is a process node, and it describes in the paragraph that the target method called only itself direct recursively, and how many times. The next node it returns in the graph is:
  - *Stop*

- **Stop:** It is a process node, but it does not contribute any text in the paragraph. It is the last node in the graph, and the traversal of the graph ends on this node.

The paragraph graph of outgoing method calls generates 97 different templates of natural language text as the paragraph graph tree is traversed from the start node to the stop node. The template does not include the data of method names and method call frequencies; they can be different.

Method saveGraph made 188 calls 100% to 14 methods. It made maximum 96 calls 51.1% to method addEdge.

(a) Outgoing method calls paragraph

Method saveGraph made 188 calls to 14 methods. It made maximum 96 calls to method addEdge.

(b) Outgoing method calls paragraph with process nodes color coding

Figure 4.16: Outgoing method calls paragraph example of a non-recursive method

Figure 4.16 shows an example of outgoing method calls paragraph for the method *saveGraph* (see Appendix A for details). Figure 4.16a shows the final output of paragraph with visualizations in the report. Figure 4.16b shows the color-coded text of the paragraph that shows which process nodes contributed what part of the text in the paragraph. The outgoing method calls paragraph graph diagram in Figure 4.15 shows the colors of the process nodes.

Method drawTreeMap made 1071 calls 100% out of which 56 calls 5.2% were direct recursive to itself. It made 1015 calls 94.8% to 13 methods. It made maximum 226 calls 21.1% to method getRect.

(a) Outgoing method calls paragraph

Method drawTreeMap made 1071 calls out of which 56 calls were direct recursive to itself. It made 1015 calls to 13 methods. It made maximum 226 calls to method getRect.

(b) Outgoing method calls paragraph with process nodes color coding

Figure 4.17: Outgoing method calls paragraph example of a recursive method

Figure 4.17 shows an example of outgoing method calls paragraph for the method *drawTreeMap* (see Appendix A for details). Figure 4.17a shows the final output of paragraph with visualizations in the report. Figure 4.17b shows the color-coded text of the paragraph that shows which process nodes contributed what part of the text in the

paragraph. The outgoing method calls paragraph graph diagram in Figure 4.15 shows the colors of the process nodes.

Recursion Depth

The recursion depth paragraph provides the information in natural language text with word-size visualizations about the recursion depth of the target method. The generator in the *Code Execution Reports* tool summarizes all the information related to recursion depth levels of the target method in this paragraph. It only generates the paragraph if the target method made recursive calls to itself during its execution. This paragraph answers the following questions:

- If the target method called itself recursively, then how many recursion depth levels it reached, and how many times it went on each level?
- If the target method reached any single recursion depth level maximum times, then which level, and how many times?

The generator creates the recursion depth paragraph shown in Figure 4.18 in the *Code Execution Reports* tool. The generator adds the word-size visualizations for the target method name and its recursion depth levels through the process nodes. The paragraph graph includes the following graph nodes:

- **Start:** It is a process node, but it does not contribute any text in the paragraph. It is the first node in the graph, and the traversal of the graph starts from this node. The next node it returns in the graph is:
  - *Recursion*
- **Recursion:** It is a decision node. The next node it selects in the graph is:
  - *Method name:* If the target method called itself recursively.
  - *Stop:* If the target method did not call itself recursively.
- **Method name:** It is a process node, and it adds the target method name in the paragraph text. The next node it returns in the graph is:
  - *Depth levels count*
- **Depth levels count:** It is a decision node. The next node it selects in the graph is:
  - *One depth level:* If the target method reached only one recursion depth level.

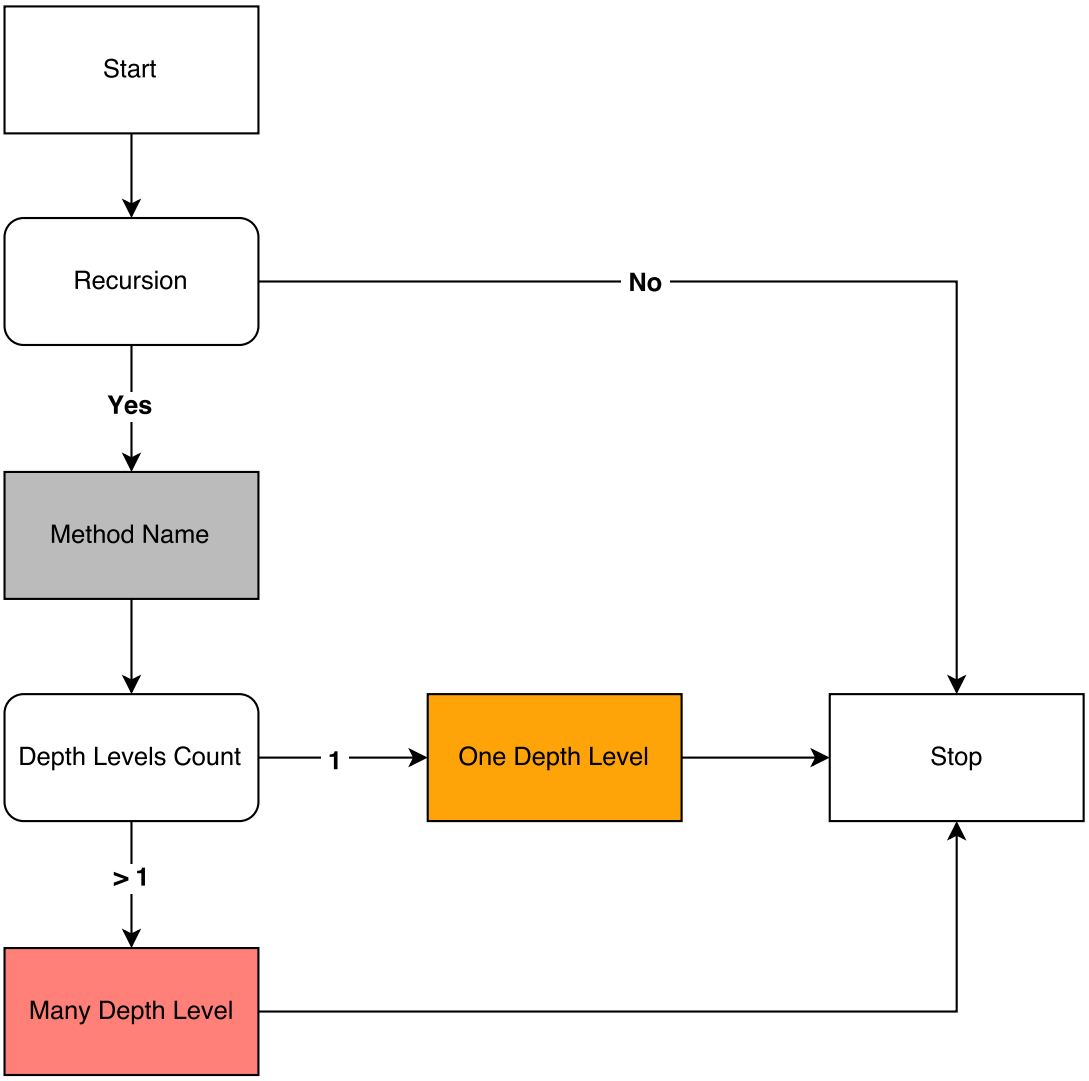


Figure 4.18: Recursion depth paragraph graph

  - *Many depth levels:* If the target method reached more than one recursion depth levels.

- **One depth level:** It is a process node, and it describes in the paragraph that the target method reached only one recursion depth level. The next node it returns in the graph is:

  - *Stop*

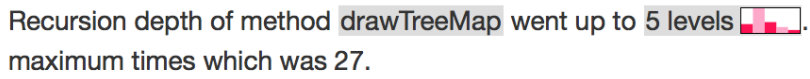
- **Many depth levels:** Many depth levels: It is a process node, and it describes in the paragraph that how many recursion depths levels the target method reached, and

how many times it reached each level, and which level the target method reached maximum times, and how many times. The next node it returns in the graph is:

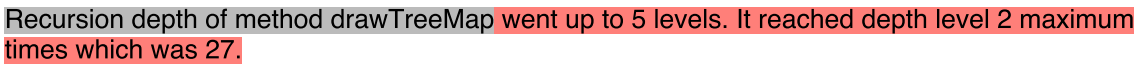
  - *Stop*

- **Stop:** It is a process node, but it does not contribute any text in the paragraph. It is the last node in the graph, and the traversal of the graph ends on this node.

The paragraph graph of recursion depth generates 4 different templates of natural language text as the paragraph graph tree is traversed from the start node to the stop node. The template does not include the data of the target method name and recursion depth levels; they can be different.



(a) Recursion depth paragraph



(b) Recursion depth paragraph with process nodes color coding

Figure 4.19: Recursion depth paragraph example of a recursive method

Figure 4.19 shows an example of recursion depth paragraph for the method *drawTreeMap* (see Appendix A for details). Figure 4.19a shows the final output of paragraph with visualizations in the report. Figure 4.19b shows the color-coded text of the paragraph that shows which process nodes contributed what part of the text in the paragraph. The recursion depth paragraph graph diagram in Figure 4.18 shows the colors of the process nodes.

Summary

The recursion paragraph provides the information in natural language text with word-size visualizations about the target method interactions with other methods as a summary. The generator in the *Code Execution Reports* tool summarizes the target method execution behavior in this paragraph. This paragraph answers the following questions:

- Did any method called the target method?
- If any of the program methods called the target method, and how many times they called the target method?

- If there is any single method that called the target method maximum times, then which method, and how many times it called the target method?
- If the single method that called the target method maximum times is the target method itself?
- Did the target method called any method?
- If the target method called any of the program methods, then how many times target method called them?
- If the target method called any single method maximum times, then how many times target method called it?
- If the target method called a single method maximum times, then is it the target method itself?
- If the target method called itself recursively, then how many recursion depth levels it reached?

The generator creates the summary paragraph shown in Figure 4.20 in the *Code Execution Reports* tool. The generator adds the word-size visualizations for the method names through the process nodes. The paragraph graph includes the following graph nodes:

- **Start:** It is a process node, but it does not contribute any text in the paragraph. It is the first node in the graph, and the traversal of the graph starts from this node. The next node it returns in the graph is:
  - *Methdod name*
- **Method name:** It is a process node, and it adds the target method name in the paragraph text. The next node it returns in the graph is:
  - *Incoming method calls count*
- **Incoming method calls count:** It is a decision node. The next node it selects in the graph is:
  - *No incoming method calls:* If no method called the target method.
  - *Recursion:* If at least one method called the target method.
- **No incoming method calls:** It is a process node, and it describes in the paragraph that no methods called the target method. The next node it returns in the graph is:
  - *Stop*

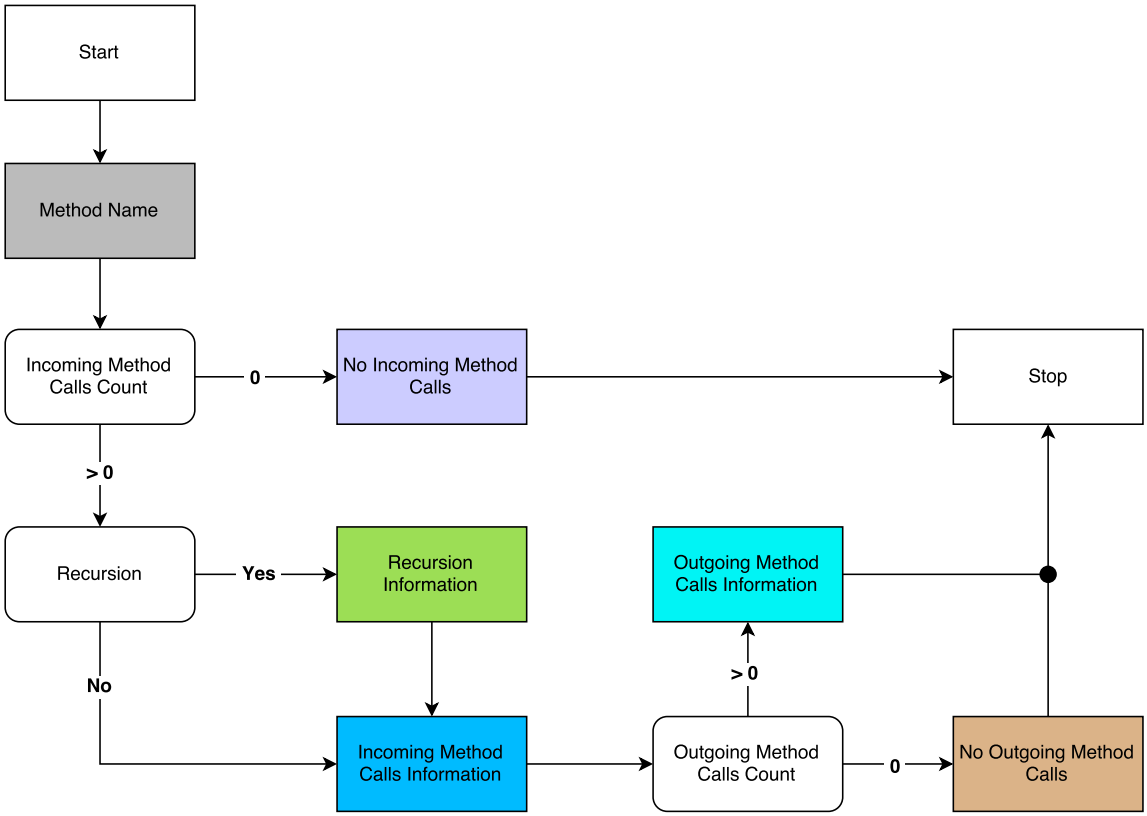


Figure 4.20: Summary paragraph graph

- **Recursion:** It is a decision node. The next node it selects in the graph is:
  - *Recursion information:* If the target method called itself recursively.
  - *Incoming method calls information:* If the target method did not call itself recursively.
- **Recursion information:** It is a process node, and it describes in the paragraph that how many times the target method called itself recursively, and how many recursion depth levels it reached. The next node it returns in the graph is:
  - *Incoming method calls information*
- **Incoming method calls information:** It is a process node, and it describes in the paragraph that how many times any of the program methods called the target method. It also describes in the paragraph about any single method that called the target method maximum times, and if that method is the target method itself. The next node it returns in the graph is:

  - *Outgoing method calls count*

- **Outgoing method calls count:** It is a decision node. The next node it selects in the graph is:
  - *Outgoing method calls information:* If the target method called at least one method.
  - *No outgoing method calls:* If the target method did not call any method.
- **Outgoing method calls information:** It is a process node, and it describes in the paragraph that how many times the target method called any of the program methods. It also describes in the paragraph about any single method that the target called method maximum times, and if that method is the target method itself. The next node it returns in the graph is:
  - *Stop*
- **No outgoing method calls:** It is a process node, and it describes in the paragraph that the target method did not call any method. The next node it returns in the graph is:
  - *Stop*
- **Stop:** It is a process node, but it does not contribute any text in the paragraph. It is the last node in the graph, and the traversal of the graph ends on this node.

The paragraph graph of summary generates 53 different templates of natural language text as the paragraph graph tree is traversed from the start node to the stop node. The template does not include the data of the method names, frequencies and recursion depth levels; they can be different.

Method saveGraph was called total 4 times with maximum calls from method updateGraph. It made total 188 outgoing calls with maximum calls to method addEdge.

(a) Summary paragraph

Method saveGraph was called total 4 times with maximum calls from method updateGraph. It made total 188 outgoing calls with maximum calls to method addEdge.

(b) Summary paragraph with process nodes color coding

Figure 4.21: Summary paragraph example of a non-recursive method

Figure 4.21 shows an example of summary paragraph for the method *saveGraph* (see Appendix A for details). Figure 4.21a shows the final output of paragraph with visualizations in the report. Figure 4.21b shows the color-coded text of the paragraph that shows which process nodes contributed what part of the text in the paragraph. The summary paragraph graph diagram in Figure 4.20 shows the colors of the process nodes.

Method drawTreeMap was recursively called with recursion depth of 5. It was called total 57 times with maximum calls from itself as direct recursion. It made total 1071 outgoing calls with maximum calls to method getRect.

(a) Summary paragraph

Method drawTreeMap was recursively called with recursion depth of 5. It was called total 57 times with maximum calls from itself as direct recursion. It made total 1071 outgoing calls with maximum calls to method getRect.

(b) Summary paragraph with process nodes color coding

Figure 4.22: Summary paragraph example of a recursive method

Figure 4.22 shows an example of summary paragraph for the method *drawTreeMap* (see Appendix A for details). Figure 4.22a shows the final output of paragraph with visualizations in the report. Figure 4.22b shows the color-coded text of the paragraph that shows which process nodes contributed what part of the text in the paragraph. The summary paragraph graph diagram in Figure 4.20 shows the colors of the process nodes.

4.2.3 Sections

Sections organize all the information in the report. They contain paragraphs, and the paragraphs contain natural language text with word-size visualizations. After generating the paragraphs, the generator component of the *Code Execution Reports* tool arranges all the paragraphs into two sections; the first section is summary, and the second section is method calls, which are described as follow:

- **Summary:** It is the first section in the report that includes only summary paragraph. It gives the overall summary of the target method runtime behavior before going into its detailed description.
- **Method Calls:** It is the second section in the report that includes three paragraphs in the following order:

1. Incoming method calls paragraph
2. Outgoing method calls paragraph
3. Recursion depth paragraph

It gives the details about the target method interactions with other methods during the program execution.

4.2.4 Final Report

Finally, the generator component of the *Code Execution Reports* tool generates the final report by using the *Report Building Framework*. The web-based interactive final report with word-size visualizations in natural language summarizes the behavior of a specified target method during its execution.

5 Evaluation

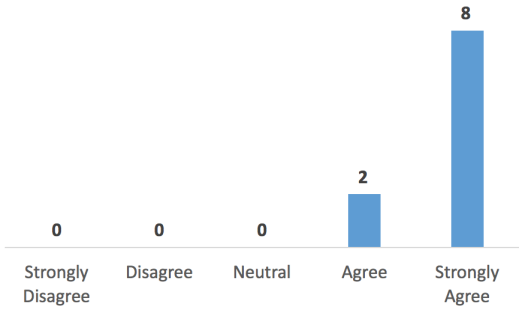
We conducted a small user study to evaluate the quality of reports that the *Code Execution Reports* tool generated. The main objective of our study was to find out the understandability of the report, and how much it is helpful in assisting the developers. Our study included 10 participants, all well experienced in object-oriented programming from either computer science or software engineering. The educational background of participants ranged from bachelor to Ph.D. students. All participants had good knowledge of human-computer interaction except one, and more than half participants were familiar with information visualization as well. 8 participants had used profilers before at least once, out of which 6 participants reported rare, and 2 participants reported regular use of profilers. All the participants who use or used profilers reported that they used them for optimization purposes; half of them also used profilers for debugging and only one participant used profilers for understanding the code as well. Table 5.1 shows the details about the participants involved in the user study.

Participant	**Education**	**Use Profilers**
P1	PhD Student	Rarely for optimization
P2	Bachelor Student	Never
P3	Bachelor	Rarely for optimization
P4	Master	Regularly for optimization and debugging
P5	Master	Regularly for optimization, debugging and understanding
P6	Master Student	Rarely for optimization and debugging
P7	Diploma Student	Rarely for optimization and debugging
P8	Master Student	Rarely for optimization
P9	Master	Rarely for optimization
P10	Diploma	Never

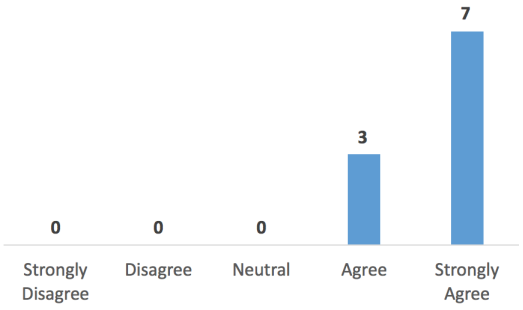
Table 5.1: User study participants details

We presented four reports of execution behavior of different methods as examples (see Appendix A for examples) generated by our tool to all participants. We asked them to read and interact with reports, then provide us feedback in an evaluation form (see

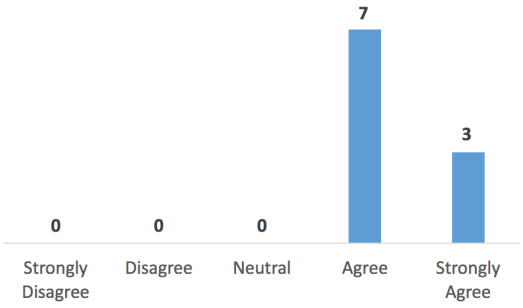
Appendix B for evaluation form). We did not provide any instructions to participants about how to interact with the reports. Evaluation feedback form asked rating scale questions on six statements using measures of agreement. It also contained an optional remarks section for participants to give their opinions about our tool and its generated reports. Figure 5.1 show all six statements with their results.



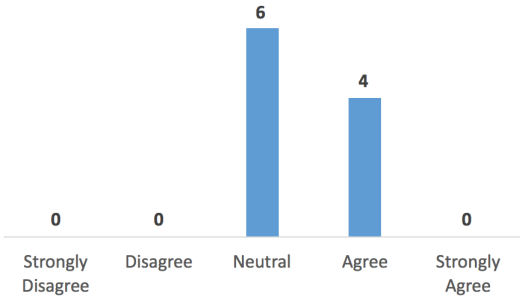
(a) Statement: "The natural language summary in the report provides a meaningful overview"



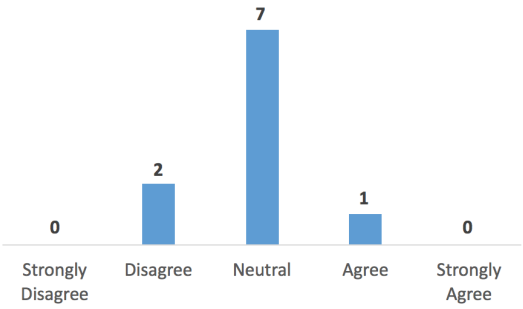
(b) Statement: "Small word-size visualizations in the report provides a good summary of quantitative information"



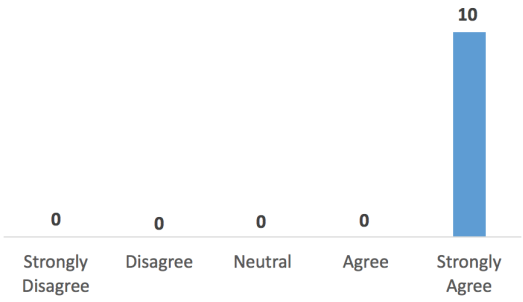
(c) Statement: "Report helps in understanding the behavior of a method"



(d) Statement: "Report helps in optimizing the method"



(e) Statement: "Report helps in debugging the method"



(f) Statement: "Report is easy to interact with and self explaining"

Figure 5.1: Evaluation results

The results of the feedback indicate that all participants agreed the summary in natural language provides a meaningful understanding. They also agreed that small word-size visualizations are summarizing the quantitative information in the reports very well. Although only one participant uses profilers for understanding the program, still all our participants agreed that reports were helpful in understanding the method. Regarding the optimization statement, all participants not responded in agreement, more than half remained neutral. Few participants disagreed that reports can assist in debugging the method, and most of them were neutral. Even though we did not provide any instructions on how to interact with the reports, but still every participant strongly agreed that reports were easy to use and self-explaining.

All participants gave positive remarks on using natural language text approach; they stated that it was effective in understanding the code. Participant P8 suggested that adding more vocabulary can improve the quality of summary. Participants P5 and P8 also appreciated the useful insights like "methods belong to the same class". Visualizations also received positive feedback from the participants; P1 and P7 liked the idea to show recursive calls using a bar chart, and P6 preferred the small visualizations over unnecessary tables. Debugging and optimization both did not receive a full agreement, but participants made some suggestions how it can be improved; participant P4 suggested that adding the consumption time information in the report can help developers in optimizing the program, and including the runtime exceptions can assist in debugging as well.

6 Conclusion

In this thesis, we have presented an approach to assist developers in understanding a method. To describe the execution behavior of a particular target method by generating a summary in natural language of its runtime activities in a visually augmented interactive report.

For this purpose, we first created the *Report Building Framework* that can generate an interactive web-based natural language report with small word-size visualizations. The framework provides the functionality of generating natural language text paragraphs, and then organizing them into sections. The framework provides three generic visualizations; progress bar, bar chart, and highlight. Small word-size visualizations are interactive, and they provide more details on demand.

Next, we developed the *Code Execution Reports* tool that uses the *Report Building Framework* to summarize the target method behavior in natural language report. Our tool profiles a specified target method, and it records all activities of the target method with other methods during the execution of the program. After profiling, it processes the recorded information, and then it summarizes the target method behavior as a combination of natural language text and word-size visualizations in the report.

Finally, we generated reports and summarized the execution behavior of different methods and conducted a small user study to evaluate our tool. We found out that our natural language summary reports provide a meaningful understanding, and small word-size visualizations among them summarize the quantitative information very well. Our participants reported that reports were easy and simple to interact, and they assisted them in understanding the method. However, reports did not provide enough information to help in debugging or optimization as they were limited to only method calls information. But adding more information in reports in future can improve this aspect of reports.

In future, reports can support more types of visualizations, and they can also extend their English vocabulary to improve the quality. They can include more information about the target method. They can also compare two different methods or two different versions of the same method.

A Examples

The *Code Execution Reports* tool generated the following four reports of execution behavior of different methods as examples for evaluation purpose.

Method *saveGraph*

Method *saveGraph* saves a directed graph. It stores the vertices and edges of the graph, and it also finds the shortest distance from the first vertex to the last vertex. It uses the JGraphT library for graph manipulation. Figure A.1 shows an example of the generated report for method *saveGraph*. The report summarizes the interactions of the method *saveGraph* with other methods during its execution.

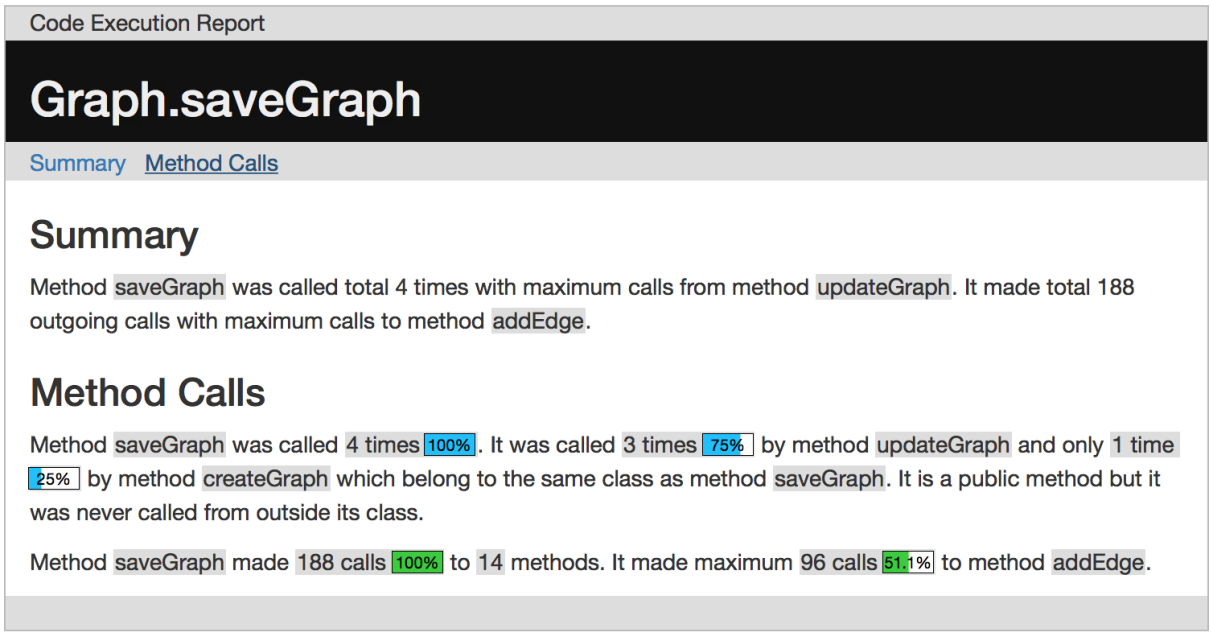


Figure A.1: Code execution report example of method *saveGraph*

Method *binarySearch*

Method *binarySearch* searches a sorted array using the binary search algorithm. It is a direct recursive method, which calls itself recursively till it finds the specified value in the array. If it successfully finds the value, then it returns the index of the value in the array, else it returns -1. Figure A.2 shows an example of the generated report for method *binarySearch*. The report summarizes the interactions of the method *binarySearch* with other methods during its execution.

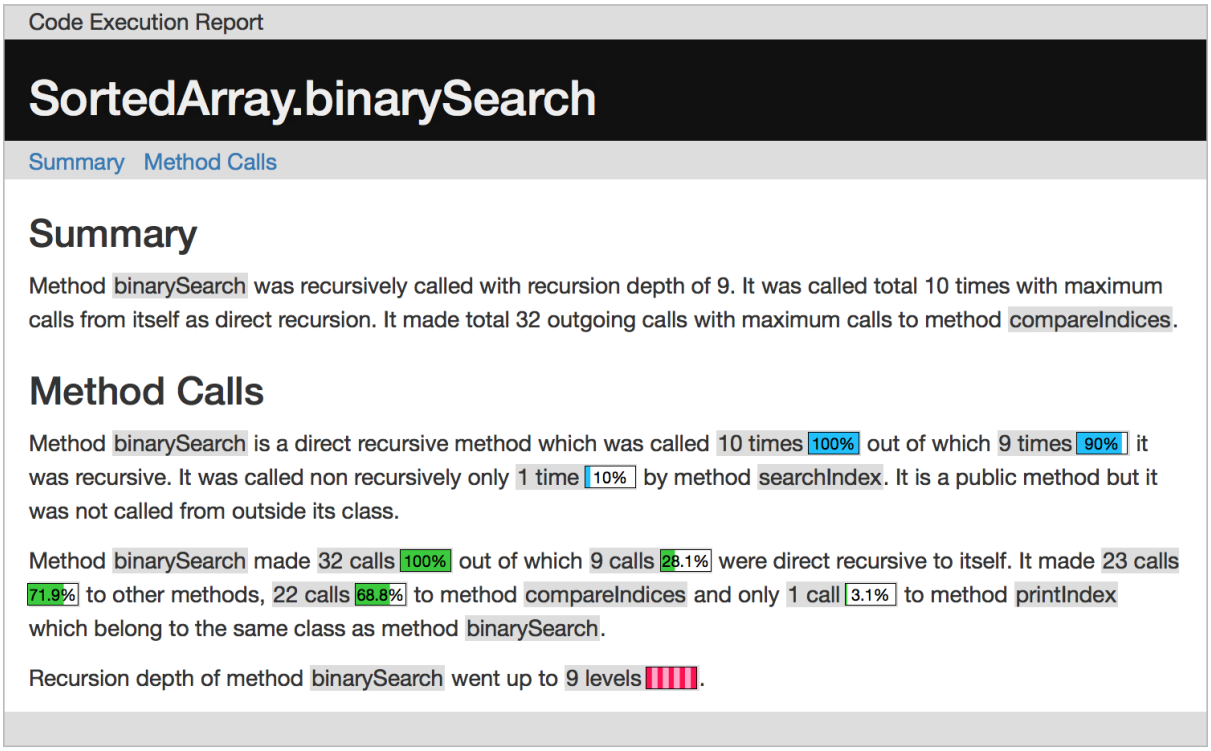


Figure A.2: Code execution report example of method *binarySearch*

Method *drawTreeMap*

Method *drawTreeMap* renders a tree map visualization. It is a direct recursive method, which calls itself recursively till it completely renders all nodes of the tree map. First, it calculates the size of the tree map rectangles from the available space for each node using X and Y coordinates, then it assigns the color to the node and renders it. Figure

A.3 shows an example of the generated report for method *drawTreeMap*. The report summarizes the interactions of the method *drawTreeMap* with other methods during its execution.

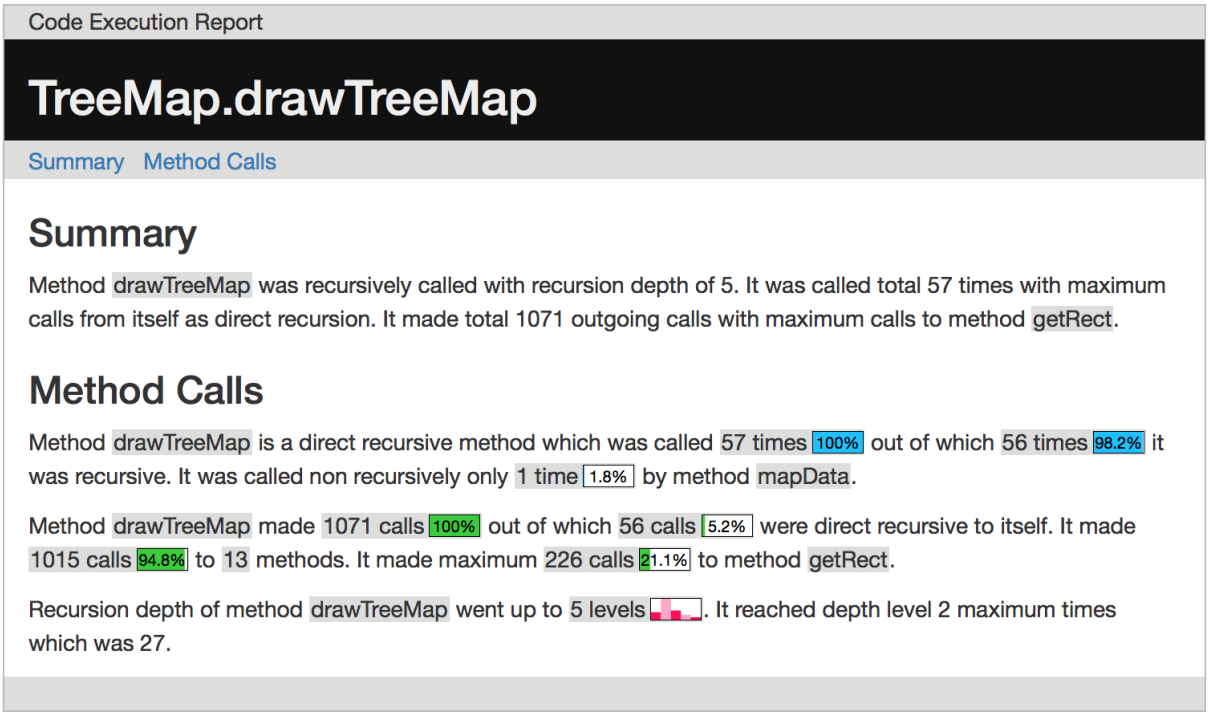


Figure A.3: Code execution report example of method *drawTreeMap*

Method *directFibonacci*

Method *directFibonacci* calculates the Fibonacci number. It is both direct and indirect recursive method, which calls itself recursively till it computes the Fibonacci number in the series at the specified index. It uses the following equation:

$$F_n = F_{n-2} + F_{n-1} \quad \text{with seed values} \quad F_1 = 1 \quad \text{and} \quad F_2 = 2$$

It uses direct recursion to calculate the *n-2* value, and it uses indirect recursion to calculate the *n-1* value. It returns the Fibonacci number after calculation. Figure A.4 shows an example of the generated report for method *directFibonacci*. The report

summarizes the interactions of the method *directFibonacci* with other methods during its execution.

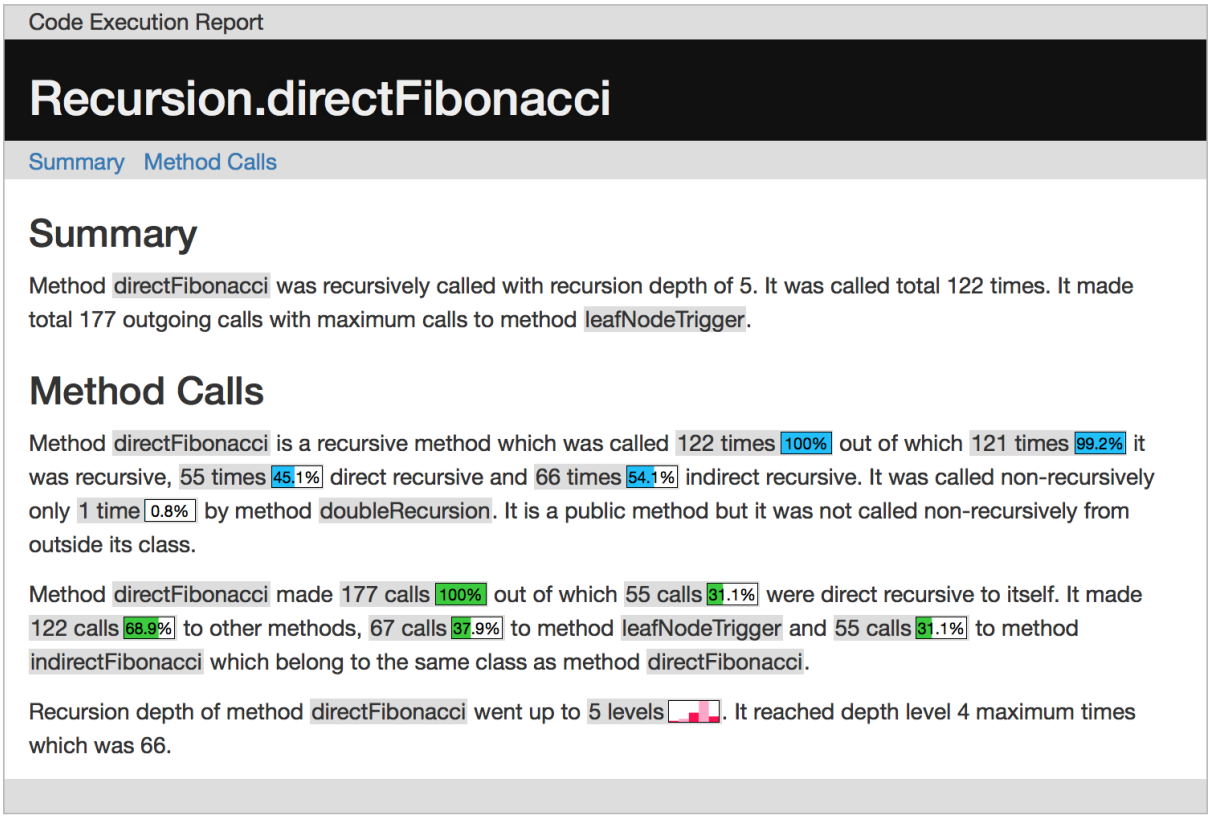


Figure A.4: Code execution report example of method *directFibonacci*

B Evaluation Form

This is an evaluation form for the *Code Execution Reports* tool. If you have not read the examples yet, please read them first before giving feedback here.

General Questions

Please provide the general information yourself.

1. What is your educational background?

   ○ Bachelor Student
   ○ Bachelor
   ○ Diploma Student
   ○ Diploma
   ○ Master Student
   ○ Master
   ○ PhD Student
   ○ Postdoc

2. What is your level of expertise in following area?

	No knowledge	Beginner	Intermediate	Expert
Object Oriented Programming	○	○	○	○
Information Visualization	○	○	○	○
Human Computer Interaction	○	○	○	○

3. Do you use profiler during programming?

- ○ Yes, regularly
- ○ Yes, rarely
- ○ No

4. If you use profiler or ever used profiler during programming then for what purpose?

- □ Optimization
- □ Debugging
- □ Understanding
- □ Never user profiler

Report Feedback Questions

Please give your opinion about each statement, and also provide any remarks after that if you have any.

5. Statement “The natural language summary in the report provides a meaningful overview” ?

Strongly Disagree ○ ○ ○ ○ ○ Strongly Agree

6. Statement “Small word-size visualizations in the report provides a good summary of quantitative information” ?

Strongly Disagree ○ ○ ○ ○ ○ Strongly Agree

7. Statement “Report helps in understanding the behavior” ?

Strongly Disagree ○ ○ ○ ○ ○ Strongly Agree

8. Statement “Report helps in optimizing the method” ?

Strongly Disagree ○ ○ ○ ○ ○ Strongly Agree

9. Statement “Report helps in debugging the method” ?

Strongly Disagree ○ ○ ○ ○ ○ Strongly Agree

10. Statement “Report is easy to interact with and self” ?

Strongly Disagree ○ ○ ○ ○ ○ Strongly Agree

11. Please provide remarks:

Bibliography

[BK11] W. Bi, J. T. Kwok. “Multi-Label Classification on Tree- and DAG-Structured Hierarchies.” In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 17–24 (cit. on p. 16).

[BL94] T. Ball, J. R. Larus. “Optimally Profiling and Tracing Programs.” In: *ACM Trans. Program. Lang. Syst.* 16.4 (July 1994), pp. 1319–1360 (cit. on p. 1).

[BMBD15] S. Baltes, O. Moseler, F. Beck, S. Diehl. “Navigate, Understand, Communicate: How Developers Locate Performance Bugs.” In: *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*. IEEE. 2015, pp. 1–10 (cit. on p. 5).

[BMDR13] F. Beck, O. Moseler, S. Diehl, G. D. Rey. “In situ understanding of performance bottlenecks through visually augmented code.” In: *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE. 2013, pp. 63–72 (cit. on p. 5).

[Bre94] C. A. Brewer. “Color use guidelines for mapping and visualization.” In: *Visualization in modern cartography* 2 (1994), pp. 123–148 (cit. on p. 26).

[Chi98] S. Chiba. “Javassist—a reflection-based programming wizard for Java.” In: *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*. Vol. 174. 1998 (cit. on p. 21).

[Cle93] W. S. Cleveland. *Visualizing data*. Hobart Press, 1993 (cit. on p. 26).

[Die07] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., 2007 (cit. on p. 2).

[Dmi04] M. Dmitriev. “Selective profiling of Java applications using dynamic bytecode instrumentation.” In: *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE. 2004, pp. 141–150 (cit. on p. 22).

[EP08] B. Eilam, Y. Poyas. “Learning with multiple representations: Extending multimedia learning beyond the lab.” In: *Learning and Instruction* 18.4 (2008), pp. 368–378 (cit. on p. 2).

[Fai06] T. Faison. *Event-Based Programming*. Springer, 2006 (cit. on p. 22).

[HAMM10] S. Haiduc, J. Aponte, L. Moreno, A. Marcus. “On the use of automated text summarization techniques for summarizing source code.” In: *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE. 2010, pp. 35–44 (cit. on p. 5).

[HIC10] M. Harward, W. Irwin, N. Churcher. “In situ software visualisation.” In: *Software Engineering Conference (ASWEC), 2010 21st Australian*. IEEE. 2010, pp. 171–180 (cit. on p. 5).

[LY99] T. Lindholm, F. Yellin. *Java Virtual Machine Specification*. 2nd. Addison-Wesley Longman Publishing Co., Inc., 1999 (cit. on p. 22).

[Man98] B. Manaris. “Natural language processing: A human-computer interaction perspective.” In: *Advances in Computers* 47 (1998), pp. 1–66 (cit. on p. 1).

[MAS+13] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, K. Vijay-Shanker. “Automatic generation of natural language summaries for java classes.” In: *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE. 2013, pp. 23–32 (cit. on p. 5).

[MM14] P. W. McBurney, C. McMillan. “Automatic documentation generation via source code summarization of method context.” In: *Proceedings of the 22nd International Conference on Program Comprehension*. ACM. 2014, pp. 279–290 (cit. on p. 5).

[Mor82] B. M. E. Moret. “Decision Trees and Diagrams.” In: *ACM Comput. Surv.* 14.4 (Dec. 1982), pp. 593–623 (cit. on p. 16).

[Mye85] B. A. Myers. “The Importance of Percent-done Progress Indicators for Computer-human Interfaces.” In: *SIGCHI Bull.* 16.4 (Apr. 1985), pp. 11–17 (cit. on p. 8).

[OH06] K. O’Hair, J. J. Heiss. “The JVM tool interface (JVM TI): how VM agents work.” In: *Web page, Dec* (2006) (cit. on p. 22).

[Oli07] R. Olivas. “Decision trees: a primer for decision-making professionals.” In: *Rev* 5 (2007), pp. 04–05 (cit. on p. 16).

[RMM10] S. Rastkar, G. C. Murphy, G. Murray. “Summarizing software artifacts: a case study of bug reports.” In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM. 2010, pp. 505–514 (cit. on p. 5).

[SHM+10] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, K. Vijay-Shanker. “Towards Automatically Generating Summary Comments for Java Methods.” In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ASE ’10. Antwerp, Belgium: ACM, 2010, pp. 43–52 (cit. on p. 5).

[TG83] E. R. Tufte, P. Graves-Morris. *The visual display of quantitative information*. Vol. 2. 9. Graphics press Cheshire, CT, 1983 (cit. on p. 1).

[Tuf06] E. R. Tufte. *Beautiful Evidence*. Graphics Press, 2006 (cit. on p. 2).

[VV95] A. Von Mayrhauser, A. M. Vans. "Program comprehension during software maintenance and evolution." In: *Computer* 28.8 (1995), pp. 44–55 (cit. on p. 1).

[YKSJ08] J. S. Yi, Y.-a. Kang, J. T. Stasko, J. A. Jacko. "Understanding and Characterizing Insights: How Do People Gain Insights Using Information Visualization?" In: *Proceedings of the 2008 Workshop on Beyond Time and Errors: Novel Evaluation Methods for Information Visualization*. ACM, 2008, 4:1–4:6 (cit. on p. 1).

[YR13] A. T. Ying, M. P. Robillard. "Code fragment summarization." In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 655–658 (cit. on p. 5).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature