JCSD: Visual Support for Understanding Code Control Structure

Ahmad Jbara[1,2] Dror G. Feitelson[2]
[1]School of Mathematics and Computer Science
Netanya Academic College, 42100 Netanya, Israel
[2]School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

ABSTRACT

Program comprehension is a vital mental process in any maintenance activity. It becomes decisive as functions get larger. Such functions are burdened with very many programming constructs as lines of code (LOC) strongly correlate with the McCabe's cyclomatic complexity (MCC). This makes it hard to capture the whole code of such functions and as a result hinders grasping their structural properties that might be essential for maintenance.

Program visualization is known as a key solution that assists in comprehending complex systems. As a matter of fact we have shown, in a recent work, that control structure diagrams (CSD) could be useful to better understand and discover structural properties of such functions. For example, we found that the code regularity property, and even cloning, can be easily identified by CSDs.

This paper presents JCSD, which is an Eclipse plug-in that implements CSD diagrams for Java methods. In particular it visualizes the control structure and nesting of a Java method, and by this it easily conveys structural characteristics of the code to the programmer and helps him to better understand and refactor.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*

General Terms

Design, Human Factors

Keywords

Visualization, code regularity, code complexity, MCC, LOC

1. INTRODUCTION

Program visualization (PV) is one aspect of Software visualization (SV). It has been defined as "the program is specified in a conventional, textual manner, and the graphics is used to illustrate some aspect of the program or its run-time execution" [10].

Software visualization has been around for a long time to facilitate both the human understanding and effective use of computer software [12]. In particular, by providing visual representation of the code (program visualization) programmers can handle and understand complex software more easily.

The comprehension of a given program becomes crucial as programs get larger. One of the key solution to this is better visualization [7]. This insight is consistent with our results about the effectiveness of using visualization to explore interesting properties of very long functions mainly in the Linux kernel and in many other systems [6, 5]. Our criteria for selecting long functions was those with high (higher than twice the highest threshold ever suggested) McCabe's cyclomatic complexity (MCC) as it is well known that MCC has a strong correlation with lines of code (LOC) [4]. Therefore, functions with high-MCC values are not only very long but also burdened with very many control constructs.

This makes it hard to capture the whole code of such functions as it spans over many pages and studying their structural properties is also not easy, in particular when such properties are scattered across the long code yet are related.

To cope with this we suggested to visualize such functions focusing on their structure and therefore we presented the control structure diagram (CSD) [6, 5]. In particular it was applied on high-MCC functions taken from the Linux kernel.

We showed that the high MCC does not really reflect effective complexity as there are functions with much higher MCC values but they are still well structured and easy to understand and handle.

This diagram has proved to be very useful as we have used it to examine the structure of these long functions. We identified regular code segments within them. Regularity in code means that code segments are repeated many times in the same function usually in a consecutive manner, but there are cases where instances appear separately. Such structural property is easily identified when the functions are visualized, especially when the block sizes are reflected in the diagram as it is the case in CSD.

Moreover, we conducted an experiment to check correlation between regularity and perceived complexity. In this experiment we got better correlation when the functions were presented to the subjects using CSD rather than the code listing.

To summarize, using CSD diagrams we realized that long functions that are supposed to be hard (according to traditional metrics) are actually well structured and easy to handle.

These promising results lead to the thought of integrating CSDs in a development environment. Our efforts produced an Eclipse plug-in called JCSD which is used to create CSD diagram for Java methods.

The JCSD tool is available for downloading and more details at URL http://www.cs.huji.ac.il/%7eahmadjbara/jcsd/jcsd.html.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICPC '14, June 2-3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06 ...$15.00.

2. THE JCSD

2.1 Design and Implementation

JCSD is a plug-in that has been developed for the Eclipse IDE for visualizing the control structure and nesting of Java methods. It is an implementation of our earlier study about CSD diagrams [6, 5]. In its current version the plug-in has one simple use case: *whenever a programmer clicks a method's name in the project explorer window of Eclipse it produces a control structure diagram (CSD) in a separate resizable window*. Figure 1 shows an example of a method and its CSD diagram.

The view window of the diagram is a typical Eclipse inner window. It is movable and resizable. It enables the programmer to move it and position it vertically or horizontally. Moreover, in cases the function is extremely long there is a possibility to resize the window which automatically scales the diagram while resizing.

The general process of creating the CSD diagram is composed of three basic stages. Initially, the source code of the method is preprocessed: comments are removed, braces are added as required, and line breaks are added, if needed, after opening and closing braces and semicolon.

After preprocessing, a tree is built where nodes represent constructs. Each node contains the construct type it represents and its block size.

On the basis of the tree built, the CSD is drawn and the tool listens to the *resize* event to enable future scaling of the diagram. Each time the *resize* event is raised the diagram is redrawn on the basis of the same tree.

2.2 Control Structure Diagram - CSD

As we stated earlier, CSD is a visualization of the control structure and nesting of a Java method. Three design criteria were considered when we designed this diagram:

- Capturing the whole code at once as much as possible.
- Reflecting the size of the different blocks in the code.
- Identifying the different types of control flow constructs.

To meet the first criterion we visualize the structure only by considering constructs and nesting while discarding simple statements. For the second criterion we allocate more space on the diagram for larger blocks. In particular we measure the size of each block and scale its representation in the diagram accordingly. For the last criterion we provide different geometric shapes and colors for different types of constructs. For example, the *if* statement is represented by a yellow trapezoid, and the wider the trapezoid the larger the block it controls in the code. Figure 1 shows an example of a Java method along with its CSD produced by the JCSD tool.

The CSD diagram is structured as follows. At the top appears the legend that maps between the different constructs and their geometric shapes. The whole function is denoted by a bar, which contains its name, underneath the legend. The content of the function is represented in the diagram from left to right while nesting is reflected by deeper levels. Moreover, empty space between the geometric shapes represents uncontrolled lines of code between controlled blocks in the code.

For example, Figure 1 visualizes a method called *nextGen*. This method receives a board representing a generation of *Conway's Game of Life* and it creates the next generation. This method, at its top level, is composed of three blocks. The first block encompasses the uncontrolled lines 3 and 4. The second block starts at line 6 and ends at 11, while the third one spans the lines 13 to 29. These three blocks are represented in the CSD diagram at the first level right underneath the function bar. The first block is represented by a proportional empty space at the left. Next to it is the second block which is represented by a small ellipse, and the third block is represented by a larger ellipse as it contains more lines in the original code than the previous block.

Levels in the diagram reflect nesting in the code. For example, the leftmost yellow trapezoid in level 3 in the diagram represents the *if* statement in line 6 in the listing which is nested in two *for* loops.

2.3 Analysis of Example Usage

To illustrate our tool we used the *jEdit* open source project which is a programmer's text editor written in the Java language. We applied our tool on the *getSize* method taken from the *ExtendedGridLayout* class of *jEdit*.

Figure 2 shows the CSD of this method while its listing is shown at URL http://www.cs.huji.ac.il/%7eahmadjbara/jcsd/getSize.java as it is quite long. Moreover, Table 1 shows its common metrics values.

According to the traditional metrics used the *getSize* method is supposed to be hard for understanding. It has 348 lines of code spanning about 6 pages. This is what a programmer really sees and it is not easy to capture the whole structure of methods with such size. However, it is easy by using a CSD.

When examining its control flow complexity (measured by MCC) we see that it is on the highest threshold ever given. Moreover, when considering its lines of code metric together with its MCC we recognize that the average size of its control blocks is about 6 lines, therefore the method is not purposelessly long but very many control constructs are interwoven within its lines. This could be easily observed by the diagram without considering complexity metrics.

The nesting metrics also indicate that this method is not easy to handle. The average nesting of all its lines is 1.3 which means that on average every line is nested. However, one might think that there is one control construct that encompasses all others and this metric does not really reflect nesting. Again, the CSD helps to identify that. In our example, this is not the case.

Based on the traditional metrics and the method's listing this method is portrayed as not easy to understand. However, examining its CSD diagram reveals a different picture. The regularity property seems to be relatively dominant as there are two large code segments (framed in Figure 2) that are likely similar. Moreover, within these two framed segments there are four repeated sub trees. The first instance appears at the left of the first frame while the second instance appears at the end of the first frame. The third and fourth instances appear in the second framed segment. These two framed segments appear at lines 60 and 173 of their method's listing at the URL presented earlier.

The fact that a code segment is totally repeated helps in investing less efforts in understanding its instances and may be an indicator for the need of refactoring.

3. RELATED WORK

One of the key solutions to code comprehension is visualization, in particular when programs get larger [7]. Nassi and Shneiderman suggested a visual model by proposing a flowchart language which prevents unrestricted transfers of control and supports structured programming [11]. In their model the details of the code are reflected as the constructs and their conditions appear within the flowchart. This might make the control over large programs difficult. Trying to grasp structural properties does not necessarily need the examination of the details. Moreover, the size of the differ-

Metric name	Description	Value
LLOC	Number of logical lines of code excluding blank lines and comments.	307
PLOC	Number of physical lines of code.	348
MCC	McCabe's cyclomatic complexity.	50
Max nesting	Max nesting within function.	3
Average nesting	Average nesting of all logical lines of code.	1.3

Table 1: Typical metrics values of the *getSize* method.

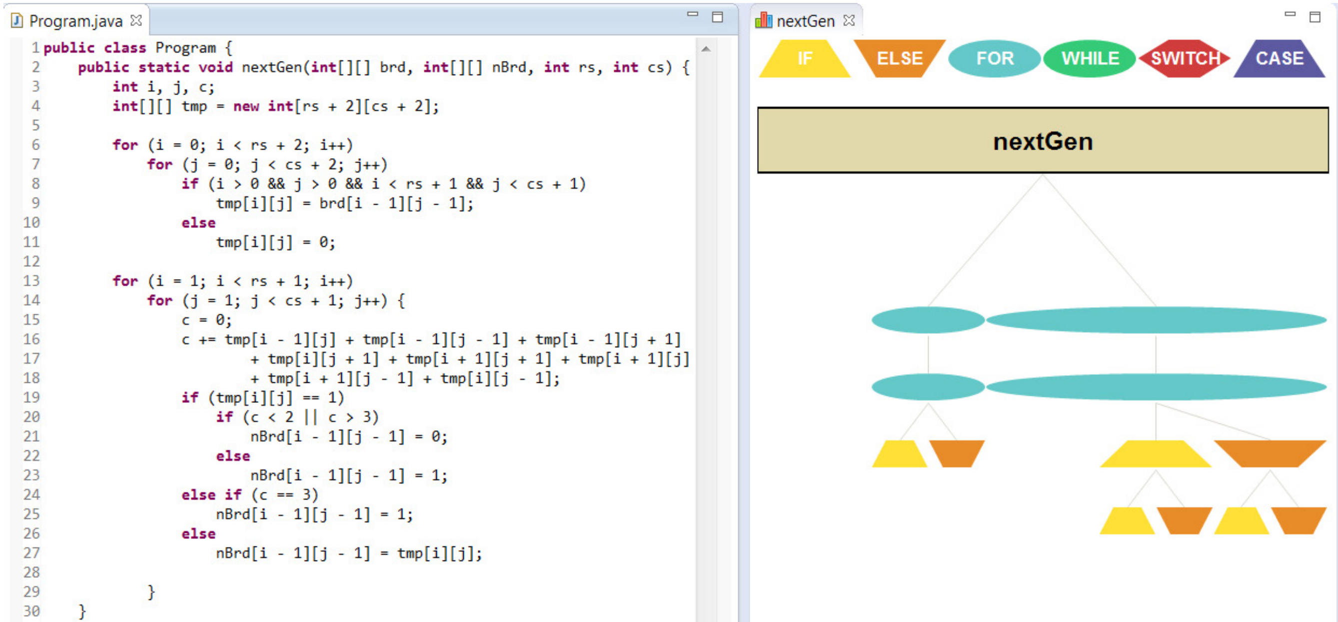

Figure 1: The control structure diagram (CSD) for the *nextGen* method of the Conway's game of life.

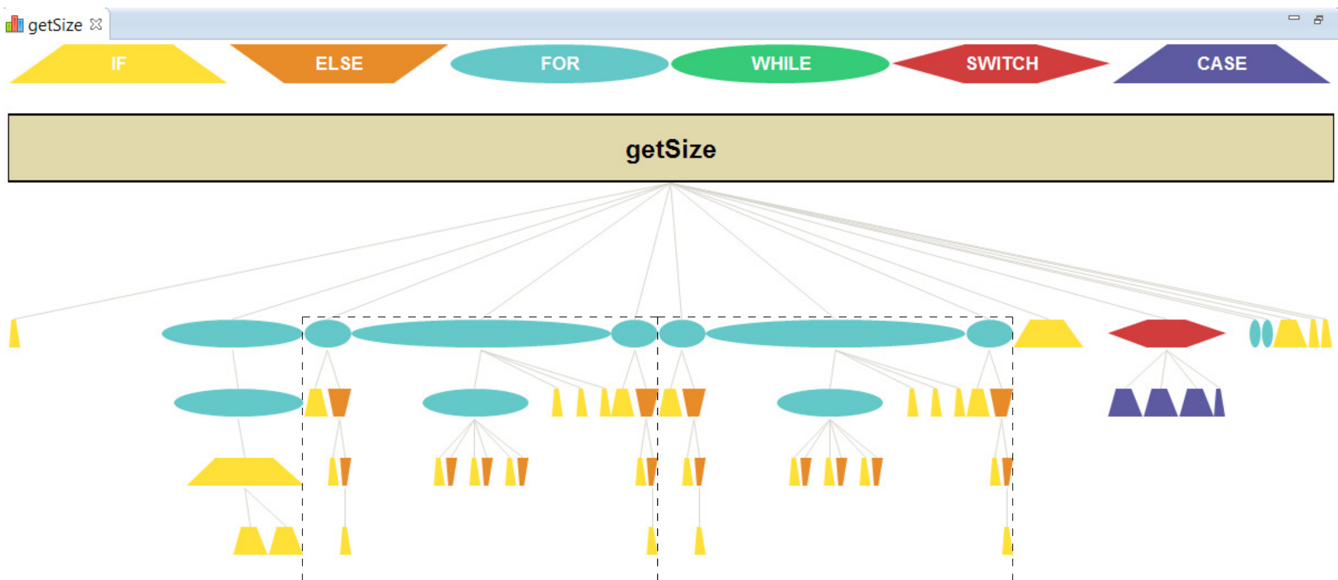

Figure 2: The control structure diagram (CSD) for the *getSize* method of the *ExtendedGridLayout* class from the *jEdit* project. Note the similarity between the structure of the framed sub trees.

ent blocks that are represented is not reflected, therefore makes it harder to capture the structure of the program.

Another work that investigates control structure is a work by Cross et al. where they present a Control Structure Diagram (CSD) to clearly depict the control constructs and the control flow at all levels of program abstraction [2]. Except of its name, it is totally different from our diagram. This diagram is superimposed upon the source code which means that the programmer is obliged to see all the code, and therefore misses the power of abstraction as he sees the details. Moreover, a diagram that reflects every line in the code would experience some difficulty in visualizing long methods which are the real challenge. In addition, their diagram is not aware to the size of the blocks that are controlled by the different constructs.

Eick et al. suggested *SeeSoft* [3, 1] where a line of code is represented by a colored line (each construct type is colored differently) with a height of one pixel and its length reflects the length of the code line. The structure of the code is reflected by preserving indentation.

Marcus et al. presented *sv3D* [9, 8]. This visualization technique is based on *SeeSoft* and added a third dimension where each code line is mapped to a rectangular cuboid.

4. FUTURE WORK

One avenue is to empirically evaluate our tool to provide an evidence of its viability in typical comprehension tasks.

Another direction is enhancing it by adding more features to help discovering more interesting structural properties. In particular, making JCSD an interactive tool. For example, it would be helpful to show the code of a specific block when the programmer's mouse hovers on its shape in the diagram, or adding a feature that enables examining simultaneously the code segments of two sub trees so the programmer could compare between them for cloning or regularity detection.

Moreover, implementing CSD for other languages and IDEs would expose it for more communities and bring more feedback. In particular, migration to other languages would be easy as the control constructs and the means for formatting and layout building are shared between different languages.

Last interesting direction is implementation of this idea as a web-based tool. The purpose is to enable programmers who search for code segments in the Internet to be able to examine its structure before copying it to his development environment.

Acknowledgments

Thanks to Elinor Alpay, Daniel Shragai, and Chen Shabo from the Computer Science Department of the Netanya Academic College for their active involvement in the implementation phase of this tool as a part of their final project of their Bachelor studies. This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

5. REFERENCES

[1] T. Ball and S. G. Eick, "*Software visualization in the large*". *Computer* **29(4)**, pp. 33–43, Apr 1996, doi:10.1109/2.488299. URL http://dx.doi.org/10.1109/2.488299

[2] I. Cross, J.H. and S. Sheppard, "*The control structure diagram*". In *Computers and Communications, 1988. Conference Proceedings., Seventh Annual International Phoenix Conference on*, pp. 274–278, Mar 1988, doi:10.1109/PCCC.1988.10084.

[3] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr., "*Seesoft-a tool for visualizing line oriented software statistics*". *IEEE Trans. Softw. Eng.* **18(11)**, pp. 957–968, Nov 1992, doi:10.1109/32.177365. URL http://dx.doi.org/10.1109/32.177365

[4] I. Herraiz and A. E. Hassan, "*Beyond lines of code: Do we need more complexity metrics?*" In *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson (eds.), pp. 125–141, O'Reilly Media Inc., 2011.

[5] A. Jbara, A. Matan, and D. G. Feitelson, "*High-MCC functions in the Linux kernel*". In *Proceedings of the 20th IEEE International Conference on Program Comprehension, ICPC 2012.*, Jun 2012.

[6] A. Jbara, A. Matan, and D. G. Feitelson, "*High-MCC functions in the Linux kernel*". *Empirical Softw. Eng.* 2013, doi:10.1007/s10664-013-9275-7. Accepted for publication.

[7] F. Lemieux and M. Salois, "*Visualization techniques for program comprehensiona literature review*". In *Proceedings of the 2006 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the Fifth SoMeT_06*, pp. 22–47, IOS Press, Amsterdam, The Netherlands, The Netherlands, 2006, ISBN 1-58603-673-4.

[8] A. Marcus, L. Feng, and J. Maletic, "*Comprehension of software analysis data using 3d visualization*". In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pp. 105–114, May 2003, doi:10.1109/WPC.2003.1199194.

[9] A. Marcus, L. Feng, and J. I. Maletic, "*3d representations for software visualization*". In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pp. 27–ff, ACM, New York, NY, USA, 2003, ISBN 1-58113-642-0, doi:10.1145/774833.774837. URL http://doi.acm.org/10.1145/774833.774837

[10] B. A. Myers, "*Taxonomies of visual programming and program visualization*". *J. Vis. Lang. Comput.* **1(1)**, pp. 97–123, Mar 1990, doi:10.1016/S1045-926X(05)80036-9.

[11] I. Nassi and B. Shneiderman, "*Flowchart techniques for structured programming*". *SIGPLAN Not.* **8(8)**, pp. 12–26, Aug 1973, doi:10.1145/953349.953350. URL http://doi.acm.org/10.1145/953349.953350

[12] B. A. Price, R. M. Baecker, and I. S. Small, "*A principled taxonomy of software visualization*". *Journal of Visual Languages & Computing* **4(3)**, pp. 211 – 266, 1993, doi:http://dx.doi.org/10.1006/jvlc.1993.1015.