International Journal of Engineering, Mathematical and Physical Sciences
ISSN: 2517-9934
Vol:5, No:3, 2011

Visualization of Code Clone Detection Results and the Implementation with Structured Data

Kazuaki Maeda

Abstract—This paper describes a code clone visualization method, called FC graph, and the implementation issues. Code clone detection tools usually show the results in a textual representation. If the results are large, it makes a problem to software maintainers with understanding them. One of the approaches to overcome the situation is visualization of code clone detection results. A scatter plot is a popular approach to the visualization. However, it represents only one-to-one correspondence and it is difficult to find correspondence of code clones over multiple files. FC graph represents correspondence among files, code clones and packages in Java. All nodes in FC graph are positioned using force-directed graph layout, which is dynamically calculated to adjust the distances of nodes until stabilizing them. We applied FC graph to some open source programs and visualized the results. In the author's experience, FC graph is helpful to grasp correspondence of code clones over multiple files and also code clones with in a file.

Keywords—code clone detection, program comprehension, software maintenance, visualization

I. Introduction

In developing programs, "copy and paste" technique is one of the simple methods to reuse source code. The "copy and paste" technique in the context of program development is called *code cloning*, and the copied and pasted portion of source code is called *code clone*. If a portion of source code is already tested, it contains fewer bugs than source code that is written from scratch. This is because developers use code cloning to reuse source code.

Many studies have been discussed code clone detection. These studies show that a significant percentage of source code contains code clones. One of the works shows that 19% of the source code is cloned in the complete source code of the X Window System [1]. Another work shows that the average percentage of code clones is 12.7% of all subsystems [2]. In an extreme case, the average percentage of code clones is 59% [3].

The code clones can be troublesome during program maintenance. If an error to be fixed is found in the original source code, all the code clones must be examined and most of the code clones should be fixed. However, the presence of code clones is usually not documented so that they must be manually detected and then fixed.

Since 1990s, many researchers have been investigating development of code clone detection tools [1]–[6]. Typical tools for detecting code clones usually show the locations with the source file names containing code clones as results. The results are typically written in a textual representation. If source code is large and the code clones are reported in the textual representation, it makes a problem to software maintainers with understanding the reports. One of the approaches to overcome the situation is visualization of code clone detection results.

Generally, it is valuable to visualize, manipulate and rearrange data in an interactive graphic. The users become not just passive observers but participants in the interaction. There are many tools for software developers to draw diagrams. Developers can navigate the written diagrams in the interactive graphic and find some suggestions using scrolling, zooming in and zooming out.

Visualization for code clone detection tools is important to grasp the distribution of code clones and to find correspondence among them. A scatter plot is a popular representation of code clone detection results. However, it represents only one-to-one correspondence so that it is difficult to find correspondence of code clones over multiple files.

This paper describes FC (File-to-Clone correspondence) graph. FC graph represents correspondence among files, code clones and packages in Java. In the implementation of FC graph, all nodes are positioned using force-directed graph layout, which is dynamically calculated to adjust the distances of nodes until stabilizing them. One of the implemented tools provides an interactive graphic on web browsers (such as Firefox [7] or Chrome [8]) to navigate code clone detection results using scrolling, zooming in and zooming out.

Most of code clone detection tools do not open the internal representation for the detected results. If the tools provide visualization capability, it is tightly coupled with code clone detection engine. Therefore, we can not use excellent visualization capability with another code clone detection tool.

The author believes that code clone detection tools should be loosely coupled the visualization capability. For this reason, graph structured data for code clone detection results was designed as a standard representation, and tools to visualize the graph structured data were implemented. In the implementation, code clone detection tools write out the results, a conversion tool reads the results and writes visualization library specific data. This paper shows that a code clone detection tool CPD [6] to detect code clones writes the results in XML and FC graph is generated from the results.

Section II describes related works of code clone detection and the visualization. Section III explains briefly about FC graph. Section IV describes visualization tools and implementation with structured data. Section V summarizes this paper.

Kazuaki Maeda is with the Department of Business Administration and Information Science, Chubu University, Kasugai, Aichi, 487-8501 Japan (e-mail: kaz@acm.org).

```
780:{
781:  private boolean select;
782:
783:  public next_line(boolean select)
784:  {
785:    this.select = select;
786:  }
787:
788:  public void actionPerformed(ActionEvent evt)
789:  {
790:    JEditTextArea textArea = getTextArea(evt);
791:    int caret = textArea.getCaretPosition();
792:    int line = textArea.getCaretLine();
793:
794:    if(line == textArea.getLineCount() - 1)

 ...................
932:{
933:  private boolean select;
934:
935:  public prev_line(boolean select)
936:  {
937:    this.select = select;
938:  }
939:
940:  public void actionPerformed(ActionEvent evt)
941:  {
942:    JEditTextArea textArea = getTextArea(evt);
943:    int caret = textArea.getCaretPosition();
944:    int line = textArea.getCaretLine();
945:
946:    if(line == 0)
```

Fig. 1. Example of code clones extracted from jEdit Syntax Package [9]

II. Related works of code clone detection and the visualization

A. Code Clone Detection

First of all, we clarify terms about code clones. The paper [4] describes that

> *A code clone* is a code portion in source files that is identical or similar to another.

Moreover, the following terms are used in this paper:

> *A code clone pair* is a pair of code portions that are identical or similar to another.
> *A code clone set* is a set of code clones paired with each other as code clone pair.

For example, two portions which is located at lines 784-792 and lines 936-944 in Fig.1 are textually identical, where the source code is extracted from InputHandler.java in jEdit Syntax Package [9]. According to the definition, the two portions are code clones and they make a code clone pair.

Many techniques related to code clone detection, are typically categorized as follows:

- Line-based approach (e.g. [1], [3])
- Token-based approach (e.g. [4], [6])
- AST-based approach (e.g. [2])
- Dependency-based approach (e.g. [5])

One of the approaches is usually embedded in a code clone detection tool. The core function of the detection is called *a code clone detection engine* in this paper.

```
<duplication lines="12" tokens="46">
<file line="783" path="jedit/InputHandler.java"/>
<file line="935" path="jedit/InputHandler.java"/>
<codefragment>
<![CDATA[
 public prev_line(boolean select)
 {
   this.select = select;
 }

 public void actionPerformed(ActionEvent evt)
 {
   JEditTextArea textArea = getTextArea(evt);
   int caret = textArea.getCaretPosition();
   int line = textArea.getCaretLine();

   if(line == 0)
]]>
</codefragment>
</duplication>
```

Fig. 2. Snippet of code clone detection result written by CPD

In the line-based approach, all lines are compared one-for-one, which provides the advantage of programming language independence. It is important issue in real applications. However, a drawback for this approach is that it ignores lexical and syntactic information in source code. If a developer changes the preferences for the locations of braces and partially executes a code formatter tool, the line-based approach fails to detect the code clones.

In the token-based approach, entire source code is scanned, a sequence of tokens is built, which are compared one-for-one. One of the token-based code clone detection tools is CPD [6]. It is an open source software and we can freely use it. If we execute CPD with a XML option to analyze the source code shown in Fig. 1, it detects that two portions are code clones and writes the output shown in Fig. 2. It is a snippet of the real XML document written by CPD. It means that the code clone is located over 12 lines of code and 46 tokens, it begins from parenthesis "(" at line number 783 and 935 in "jedit/InputHandler.java," and it ends to equals "==" at line number 794 and 946.

In the AST-based approach, syntax sensitive analysis detects code clones precisely. Generally, a compiler constructs an AST in the syntax analysis phase to represent syntactic information in the source code. By modifying the compiler or building a syntax analyzer from scratch, the AST can be derived from the source code. We can obtain more precise results related to code clones if the CFG and DFG analysis is applied to detect them. However, if the approaches are applied to other programming languages, excessive development cost will be required.

All approaches have their advantages and disadvantages. An aim of this research is to realize an independent visualization of code clone detection engines. This paper is one step toward the independence. CPD was chosen to examine it as one of code clone detection engines.

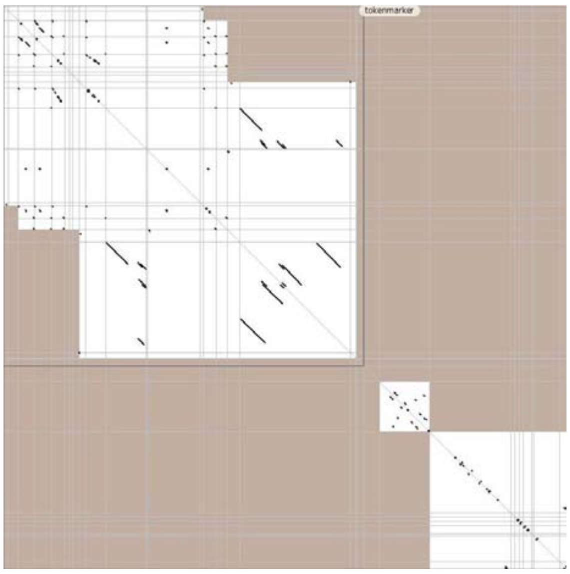

Fig. 3. Scatter plot displayed by CCFinderX

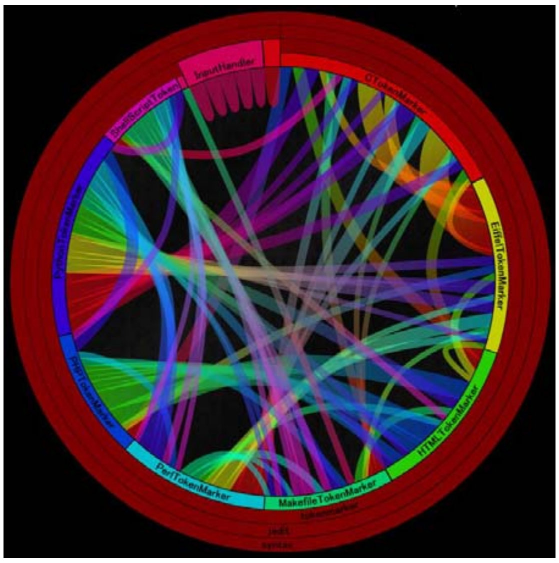

Fig. 4. Wheel displayed by Atomiq

B. *Visualization of Code Clone Detection Results*

A textual representation is a simple method of code clone detection results. However, software maintainers have a problem to understand the results if the target source code to detect code clones is large and the results are written in the textual representation. To overcome the situation, there are some visualizing approaches of code clone detection results to help software maintenance.

CCFinderX [11] uses a scatter plot to show code clone detection results. For example, Fig. 3 is the scatter plot displayed by CCFinderX after detecting code clones of jEdit Syntax Package version 2.2.2 [9]. In the scatter plot, the vertical and horizontal axes represent location on source files. Dots mean code clones which represent one-to-one correspondence. As a result, the dots are plotted symmetrically against the diagonal line.

Atomiq [12] is a commercial product and visualizes one-to-one correspondence of code clones onto a wheel in a circular layout. Fig. 4 shows an example of the wheel displayed by Atomiq. A file is represented as an arc on the wheel, and different colors distinguish between files. One-to-one correspondence of code clones is represented as connection using curved lines between portions through the interior of the wheel. The clones located in a same file are represented as an adjacent curved line so that it looks like a swelling shown in upper area of Fig.4. The clones located in other files represented as an arch with gradation between a file color and another color. The circular representation realizes more easy visualization than linear representation. We can globally grasp correspondence of code clones using the wheel.

The representation of Atomiq is just only one-to-one correspondence which is same as CCFinderX. If a portion is cloned to more than two files, we can not grasp intuitively correspondence among the code clones from the visualization displayed by CCFinderX and Atomiq.

Another approach is to visualize correspondence between code clones using undirected graph [13]. It is called a code clone graph in this paper. Let us use an example in Fig.5 which is extracted from the paper [13]. In the figure, we have four files (represented as rectangles) and eight code clone sets (represented as circles) in the files so that the code clone graph is built as shown in Fig.6. An edge between two nodes means the code clones are included in one file. For example, code clone 1 and 2 are located in a same file so that node 1 and 2 are connected. In the same way, node 2 and 4 are connected. We can get the code clone graph after repeating it to all pairs of node.

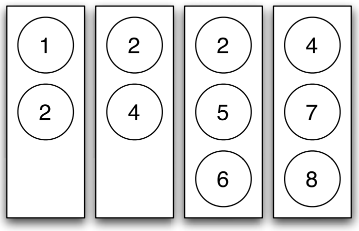

Fig. 5. Eight code clone sets and four files

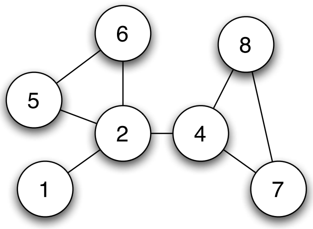

Fig. 6. Code clone graph in the case of Fig.5

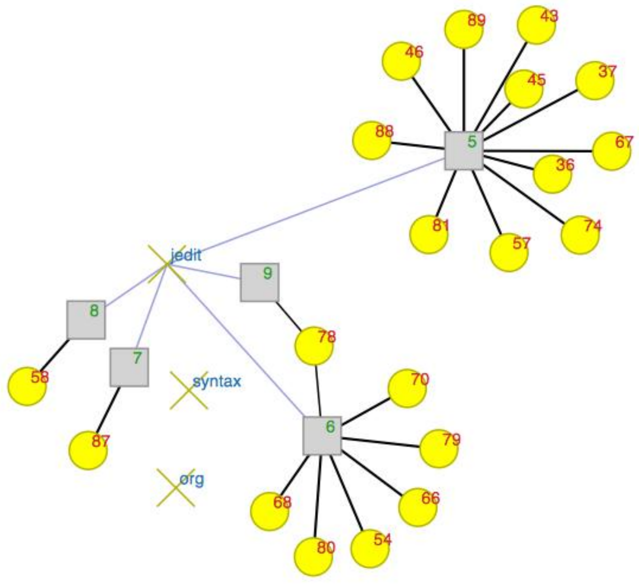

Fig. 7. Snippet of FC graph for jEdit Syntax Package version 2.2.2

III. FC GRAPH

If we use the scatter plot as code clone detection results, we can grasp distribution of code clones globally and find code clone pairs in one same file or two files. However, it is difficult to find code clone sets in multiple files. If we have a long list of code clone pairs, the wheel displayed by Atomiq helps users see one-to-one correspondence patterns in the easy way. However, because of the same reason as the scatter plot, we can not find code clone sets in multiple files. If we use the code clone graph, we can easily view correspondence of code clone pairs located in same files. However, we can not find code clone sets spread over multiple files.

From the investigation into visualization of code clone detection results, FC (File-to-Clone correspondence) graph was designed. Fig. 7 is a snippet of FC graph to show code clones detection results of jEdit Syntax Package version 2.2.2. In FC graph, there are three types of nodes, a square, a circle and a cross. A square means a file, a circle means a code clone set and a cross means a package in Java. A sequential number is assigned to a file and a code clone set.

In Fig. 7, there are five files of which numbers are 5, 6, 7, 8 and 9, and twenty code clone sets of which numbers are 36, 37, 43, 45, 46, 54, 57, 58, 66, 67, 68, 70, 74, 78, 79, 87, 80, 81, 88 and 89. An edge between a circle and a square means the code clone set exists in the file. For example, in the left side, there is an edge between a circle with number 58 and a square with number 8. It means that the code clone set 58 exists in the file 8 so that more than one code portion located in the file 8 are identical or similar to another. Moreover, there are edges among a square with number 9, a circle with number 78 and another square with number 6. It means that the code clone set 78 exists in the file 9 and the file 6 so that one code portion in a file is cloned to another file.

Coordinates of nodes in FC graph are decided using force-directed graph layout [14]. In the force-directed graph layout, edges of the graph have spring-like forces, and calculation to adjust lengths of edges keeps going until stabilizing distances of nodes. There are invisible edges between crosses, and visible edges between a cross and a square. Correspondence between packages is not important for code clone detection. To position closely files belonged in a same package, crosses are plotted in FC graph but edges between crosses are invisible.

IV. IMPLEMENTATION ISSUES OF FC GRAPH

A. Visualization Tools

There are many commercial visualization tools and open source visualization tools. Generally, the visualization tools are divided into two groups. One type of the group is a standalone tool, the other is a library-based tool.

If we use the standalone visualization tool, we can execute the tool without extensions or plug-ins, and do some works like text editors. It reads a data file written in a textual or binary format, visualizes it, and analyzes it in some techniques.

Gephi is an excellent standalone visualization tool [15], [16]. It is an open source software and it has been developed to analyze social networks and semantic Web. Gephi is based on NetBeans platform so that we can extend the capability using plug-in APIs. Cytoscape is another standalone visualization tool [17]. It has been developed to visualize molecular interaction networks and other data, but we can use it to visualize and analyze other networks than the molecular networks.

These tools are very powerful, but we decided not to use the standalone tools. We need to embed visualization function and control it in our implementation so that we decided to choose library-based tools.

One of the library-based tools is Protovis [18], [19]. It is a JavaScript library to make interactive visualization more accessible to web. Protovis realizes platform independence. If we have any web browsers containing a JavaScript processor, visualization flies over the Internet and we can see it on the web browsers. Our implemented tools have an option to generate a HTML file and graph data to display FC graph by Protovis. Fig. 7 is a snippet of a real output[1] on Firefox web browser after detecting code clones of jEdit Syntax Package version 2.2.2.

JUNG [20] and prefuse [21], [22] are Java libraries which are helpful to build Java Swing based visualization applications. The implementation work for FC graph is now going to support JUNG and prefuse.

B. Representation of Graph Structured Data in RugsOn

To build FC graph, CPD2RugsOn and RugsOn2Vis were implemented. Fig. 8 shows procedures to build FC graph from source code in Java. CPD is a code clone detection tool which writes out code clone detection results in XML. CPD2RugsOn reads the results in XML, and analyzes correspondence among files, code clones and packages in Java. After that it writes graph structured data in RugsOn [10], which was proposed by

[1]The real output is open at http://www.bais.chubu.ac.jp/˜kaz/clones/

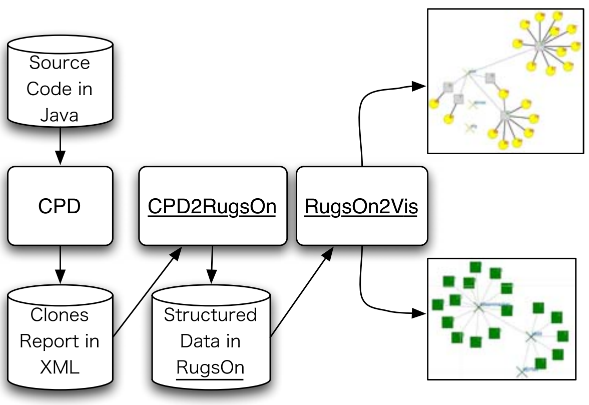

Fig. 8. Procedures to build FC graph from source code in Java

the author. RugsOn2Vis reads the graph structured data, and writes FC graph data in some data formats.

RugsOn is used to represent graph structured data. The representation in RugsOn is composed of several elements. Each element has a value and a name. For example,

```
"syntax" .nodeName
```

which represents the value as "syntax" and the name of the element as *nodeName*. The element is not only a data representation, but also it is an executable method invocation without parentheses in programming languages (e.g. Ruby). The method name is *name* and the receiver of the method is "syntax."

In RugsOn, a structure is represented using a block as an argument. For example, Fig. 9 shows that *node* has four child elements: *nodeName*, *kind*, *size* and *uuid*. In the case of the first *node* element, *nodeName* element's value is "syntax," the *kind* element's value is 1, the *size* element's value is 1, and *uuid* element's value is "`:id_0`."

An element to represent a collection can have more than one element with the same name. In Fig. 9, the *nodes* element has six child elements with the name *node*. The first *node* element has a *nodeName* element with a value "syntax," the second *node* element has a *nodeName* element with a value "jedit," the third *node* element has a *nodeName* element with a value "org." This represents a sequence of *node* elements.

If we need to represent an element linked to another element across the structure, a unique identifier, called *uuid*, is given to the element, and another element refers to the element using the identifier. Fig. 9 shows that a *uuid* element with the identifier "`:id_0`" is given to the first *node* element, and another *uuid* element with the identifier "`:id_3`" is given to the third *node* element. The *source* element in the first *link* element has a reference to the identifier "`:id_0`," and the *target* element in the first *link* element has a reference to another identifier "`:id_3`." The representation shows links across the structure so that RugsOn supports graph structured data. When a program writes graph structured data to a file and another program reads the data from the file, it reconstructs the graph structured data. RugsOn specific APIs are prepared so that the value of the *uuid* element can be freely modified.

```
cloneGraph {
  nodes {
    node {
      "syntax"  .nodeName
      1  .kind
      1  .size
      ":id_0"  .uuid
    }
    node {
      "jedit"  .nodeName
      1  .kind
      1  .size
      ":id_1"  .uuid
    }
    node {
      "org"  .nodeName
      1  .kind
      1  .size
      ":id_3"  .uuid
    }
    node {
      "."  .nodeName
      1  .kind
      1  .size
      ":id_4"  .uuid
    }
    node {
      "8"  .nodeName
      1  .kind
      1  .size
      ":id_5"  .uuid
    }
    node {
      "58"  .nodeName
      3  .kind
      1  .size
      ":id_58"  .uuid
    }
  }
  links {
    link {
      ":id_0"  .source
      ":id_3"  .target
      1  .kind
    }
    link {
      ":id_1"  .source
      ":id_0"  .target
      1  .kind
    }
    link {
      ":id_8"  .source
      ":id_1"  .target
      2  .kind
    }
    link {
      ":id_58"  .source
      ":id_8"  .target
      3  .kind
    }
  }
}
```

Fig. 9. Graph structured data for FC graph in RugsOn

RugsOn plays an important role of intermediate representation to realize visualization tool independence. In current implementation, RugsOn2Vis has only one option to generate FC graph data for Protovis. Other options will be implemented to generate the graph data for JUNG and prefuse.

V. Conclusion

This paper describes a code clone visualization method, FC graph, and the implementation issues. There are many approaches to visualization of code clone detection results such as a scatter plot and a wheel. However, it represents only one-to-one correspondence and it is difficult to find relationships of code clones over multiple files. FC graph represents correspondence among files, code clones and packages in Java. In current implementation, nodes in FC graph are positioned using force-directed graph layout, which is dynamically calculated to adjust the distances of nodes until stabilizing them. In the author's experience, FC graph is helpful to grasp correspondence of code clones over multiple files.

Graph structured data for code clone detection results was designed for loosely coupling with code clone detection engines and the visualization capability. In this paper, an open source software CPD was chosen to detect code clones. CPD writes the results in XML and FC graph is generated from the results. During generating procedures, RugsOn plays an important role of intermediate representation. Current implementation provides only one option to generate FC graph data for Protovis. Research and development about visualizing code clone detection results will continue to generate FC graph data for other libraries and to support other visualization methods. The results will be published in a future paper.

Acknowledgment

The author would like to thank Mio Andoh and Michio Takahashi for discussing FC graph and the implementation.

References

[1] Brenda .S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," Working Conference on Reverse Engineering, pp.86–95, 1995.
[2] Ira D. Baxter, Andrew Yahin, et al., "Clone Detection Using Abstract Syntax Trees," International Conference on Software Maintenance, pp.368-377, 1998.
[3] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer, "A Language Independent Approach for Detecting Duplicated Code," 15th IEEE International Conference on Software Maintenance, pp.109–118, 1999.
[4] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," IEEE Transactions on Software Engineering, vol.28, no.7, pp.654–670, 2002.
[5] Jens Krinke, "Identifying Similar Code with Program Dependence Graphs," Working Conference on Reverse Engineering, pp.301–309, 2001.
[6] PMD: Finding copied and pasted code, available from http://pmd.sourceforge.net/cpd.html.
[7] Mozilla — Firefox web browser & Thunderbird email client, available from http://www.firefox.com/.
[8] Google Chrome - Get a fast new browser. For PC, Mac, and Linux, available from http://www.google.com/chrome/.
[9] jEdit Syntax Package - Open Source syntax highlighting JavaBean, available from http://syntax.jedit.org/.
[10] Kazuaki Maeda, "Executable Representation for Structured Data Using Ruby and Scala," 10th International Symposium on Communications and Information Technologies, pp.127–132, 2010.
[11] CCFinder Official Site, available from http://www.ccfinder.net/ccfinderx.html.
[12] Atomiq : Code Similarity Finder, available from http://www.getatomiq.com/.
[13] Yoshihiko Fukushima, Raula Kula, Shinji Kawaguchi, et al., "Code Clone Graph Metrics for Detecting Diffused Code Clones," 16th Asia-Pacific Software Engineering Conference, pp.373–380, 2009.
[14] Andreas Noack, "Energy Models for Graph Clustering," Journal of Graph Algorithms and Applications, vol.11, no.2, pp.453–480, 2007.
[15] Gephi, an open source graph visualization and manipulation software, available from http://gephi.org/.
[16] Mathieu Bastian, Sebastien Heymann and Mathieu Jacomy, "Gephi: An Open Source Software for Exploring and Manipulating Networks," International AAAI Conference on Weblogs and Social Media, pp.361–362, 2009.
[17] Cytocape: An Open Source Platform for Complex-Network Analysis and Visualization, available from http://www.cytoscape.org/.
[18] Protovis, available from http://vis.stanford.edu/protovis/.
[19] Michael Bostock and Jeffrey Heer, "Protovis: A Graphical Toolkit for Visualization," IEEE Transactions on Visualization and Computer Graphics, vol.15, no.6, pp.1121–1128, 2009.
[20] JUNG - Java Universal Network/Graph Framework, available from http://jung.sourceforge.net/.
[21] prefuse — interactive information visualization toolkit, available from http://prefuse.org/.
[22] Jeffrey Heer, Stuart K. Card and James A. Landay, "prefuse: a toolkit for interactive information visualization," The SIGCHI Conference on Human Factors in Computing Systems, pp.421-430, 2005.

Kazuaki Maeda He is a professor of Department of Business Administration and Information Science at Chubu University in Japan. He is a member of ACM, IEEE, IPSJ and IEICE. His research interests are Compiler Construction, Domain Specific Languages, Object-Oriented Programming, Software Engineering and Open Source Software.