Journal of Zhejiang University SCIENCE
ISSN 1009-3095
http://www.zju.edu.cn/jzus
E-mail: jzus@zju.edu.cn

MFV-Class: a multi-faceted visualization tool of object classes*

ZHANG Zhi-meng (张志猛)†, PAN Yun-he (潘云鹤), ZHUANG Yue-ting (庄越挺)

(*Institute of Artificial Intelligence, College of Computer Science, Zhejiang University, Hangzhou 310027, China*)

†E-mail: zzmeng@sina.com

Received June 26, 2003; revision accepted Dec. 14, 2003

Abstract: Classes are key software components in an object-oriented software system. In many industrial OO software systems, there are some classes that have complicated structure and relationships. So in the processes of software maintenance, testing, software reengineering, software reuse and software restructure, it is a challenge for software engineers to understand these classes thoroughly. This paper proposes a class comprehension model based on constructivist learning theory, and implements a software visualization tool (MFV-Class) to help in the comprehension of a class. The tool provides multiple views of class to uncover manifold facets of class contents. It enables visualizing three object-oriented metrics of classes to help users focus on the understanding process. A case study was conducted to evaluate our approach and the toolkit.

Key words: Program comprehension, Reverse engineering, Software visualization, Object-oriented software metrics, Program analysis

doi:10.1631/jzus.2004.1374 **Document code:** A **CLC number:** TP311.5

INTRODUCTION

The increasingly complex and large number of software products in today's software industry makes the tasks of software maintenance, software reuse and software restructuring more and more difficult. With the popularity of the object-oriented programming paradigm, the ability to reverse engineer and understand object-oriented legacy systems has become an important problem to be addressed. In the object-oriented reverse engineering (OORE) research area, most researchers focus on the discovery of high-level design knowledge, such as the software architecture and objects relations (Zhang *et al*., 2003). This is derived from two reasons: one is that OO paradigm has more emphasis on high-level design, another is that OO design theory encourages designing small classes. Our early research work in OORE showed that there are three specific needs for a tool to support the reverse engineering and comprehension of a single class:

The first is that some gigantic classes do exist in many industrial systems, and these classes are complicated artifacts by themselves. For example: de.uni_paderborn.fujaba.basic.JavaFactory, a class in FUJABA (http://www.upb.de/cs/FUJABA/), has 122 methods, with 6000 lines source code; a class of jigsaw (http://www.w3.org/Jigsaw/), org.w3c.jigsaw. webdav.DAVFrame, has 99 methods and 25 fields, with 2700 lines source code. Obviously, the understanding method based on source code reading is not enough to comprehend here.

The second need is that classes map the key

*Project supported by the National Basic Research Program (973) of China (No. 2002CB312101), the National Natural Science Foundation of China (No. 60272031), Doctorate Research Foundation of the State Education Commission of China (No. 20010335049), Zhejiang Provincial Natural Science Foundation of China (No. ZD0212)

application domain concepts and represent the primary abstractions from an application domain in the object-oriented software. For example, most of the terms coming from the application domain are directly mapped to OO classes in the system analysis phase of the OO paradigm. And the complexity of OO software comes mostly from the complexity of the application domain (DeBaud *et al*., 1994). So understanding each class thoroughly is the basis for understanding the application domain, as well as understanding an OO software system for software engineers.

The last need is that sometimes an OO software system or a subsystem has its complicated logic or flow control in a single class, so class comprehension is the key to further understanding the software system or subsystem.

Above all, the extraction and abstraction of a class source code are absolutely necessary for object-oriented software comprehension and its reverse engineering. By now, besides one approach that focused on the "taste" comprehension of a single class through its "class blueprint" (Michele and Stephane, 2001), there are no other special CASE tools to aid this phase. In this paper, we propose a class comprehension model based on constructivist learning theory, and describe a software object visualization tool (MFV-Class) to help single class comprehension by automatically creating multi-faceted views of any single class.

The rest of this paper is structured as follows: in the next section we present the class comprehension model based on the constructivist learning theory. Section 3 describes the meta-model of class source code extraction. Section 4 illustrates the implementation details of MFV-Class. Section 5 shows some results of one case study. The last section gives our conclusion and an outlook on future work on this topic.

CLASS COMPRHENSION MODEL

There are many software comprehension models in software engineering, and these can be categorized to three ways: Brooks' (1983) top-down program comprehension strategy, a "hypothesis driven approach" in which an initially vague and general hypothesis is refined and elaborated based on information extracted from the program text and other documentation. The second is a bottom-up comprehension strategy (Pennington, 1987), an approach that works by first reading the source code and then mentally chunking the low-level software artifacts into meaningful, higher-level mental abstractions. The last approach is "as-needed" comprehension strategy (Littman *et al*., 1987), which merges the above two methods in its process by making hypothesis, exploring information, and validating hypothesis needed for the comprehension task. This as-needed comprehension strategy is the most popular method in software maintenance activities today.

The task of single class comprehension is somewhat different from understanding a software system as a whole, since a class is an artifact with some formal structure. In fact, the single class comprehension is more like a concept construction process.

According to the constructivist learning theory (Derry, 1996), constructing a concept is the core activity in human learning, recognition and comprehension. The theory includes three main points: (i) There is no knowledge independent of the meaning attributed to experience (constructed) by a learner; it shows that comprehension is a gradual assimilation process. (ii) The focus must be on the learner in thinking about learning and not on the material to be understood (This point conveys that the concrete view closed to internal structure of human knowledge is more important than source code for learner). And (iii) the exact comprehension process involved is different from learner to learner; this shows that the information should be provided in a way that best suits each learner.

Overall, class comprehension can be fostered by the use of various automatically created views of the class which reveal different aspects of the class structure and behavior. Taking into account the object-oriented characteristics implied in each class design, such as inheritance and abstraction, we propose that class comprehension should cover the

following facets:

1) Intension of a concept: understanding a class should be started from exploring its internal structure and relationship. These include data flow and control flow perspectives. The data flow view focuses on the comprehension of data elements, as opposed to the static structure of a class. Control flow view uncovers how to implement the services of a class, and this view is important for a software engineer to understand the design knowledge of responsibility assignment and class encapsulation.

2) Extension of concept: A class is not isolated from others in the OO, so understanding the context in which a class exists is important. One view is on what is the interface it provides to others, and another view is what services and data it needs from other classes.

3) Inheritance perspective: inheritance is related to the classification of concepts, and is the key feature of the OO paradigm. The usage of inheritance enhances the comprehension of complex OO software (Li *et al*., 2000). So inheritance view is valuable for understanding the class design. There are two inter communication types between a super class and a subclass: one is the directly inherited members or constructors, and the other is the invocation or message passings between them. Understanding the invocation of methods is absolutely critical to understand the class.

CLASS CODE EXTRACTING MODEL

There are two levels of components that a class code extractor can extract: one is the class member level, and the other is the statement level. For class comprehension, it is more important to get the relationship between the members of the class than the relationship between the statements of the class. Our extracting model is thus targeted at member level extraction, as illustrated in Fig.1 (which is drawn in UML class diagram style). In Fig.1, the three subtypes, "Reference_ Class", "Super_Class" and "This_Class", inherit all five fields from the abstract type "Class". "This_Class" represents the class that is to be extracted, and has an "inherence"

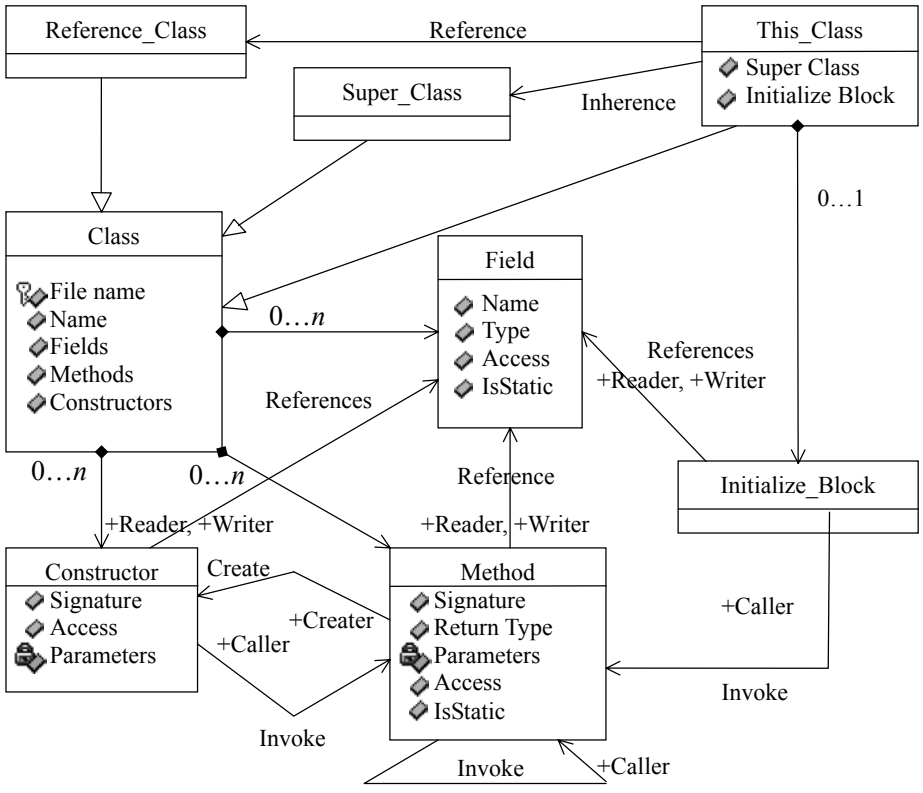


Fig.1 Extracting model of class code

relationship with "Super_Class". In Java, "Super_Class" can be a directed super class of "This_Class" or interfaces that "This_Class" have implemented. "Reference_Class" represents the classes referenced by the method code of "This_Class". "Class" can have any number of "Method", "Field", "Constructor", but only "This_Class" may have one "Initialize Block". There are varieties of relationships between members: "Method" and "Constructor" can invoke reciprocally, "Initialize Block" can invoke "Method", and all three can reference "Field" through a "Reader" role or a "Writer" role.

To show the status of each component in the "This_Class", we define three class metrics: NOC and NOLV metrics are about the methods, and NOMR is about fields. Their definitions are as follows:

1) NOC (Number Of Calls): the sum of method calls in a method body. The nested calls in code are counted separately.

2) NOLV (Number Of Local Variables): the sum of local variables in a method body; it counts the parameters of the method.

3) NOMR (Number Of Method References): the sum of the field references by methods, constructors and initialization block of "This_Class".

MFV-CLASS FRAMWORK

Context of MFV-Class

MFV-Class is developed to help user understand a complex class in an OO software system. It is a part of our OO reverse engineering environment called AUTOORE. AUTOORE is composed of an OO language parser, an extractor, a software object cluster tool, a reasoning tool and a software visualization tool. Because we have only implemented the Java parser, this tool now can only be used for Java programs.

Architecture of MFV-Class

The architecture of MFV-Class is illustrated in Fig.2, where the "Basic Class Data" comes from the Java parser in AUTOORE, which records all class names in the OO system. Such data are used by the extractor to decide whether the referenced items of the class belong to the OO system. The extractor is based on an abstract syntax tree which is parsed by the class parser. In the implementation of class parser, we used some java parse classes in the packages of sun.tools.java, sun.tools.javac and sun.tools.tree.

The visualization tool of MFV-Class is based on VGJ (http://www.eng.auburn.edu/department/cse/research/graph_drawing/) (Visualizing Graphs with Java). The graph file format in VGJ is GML (http://www.infosum.fmi.uni-passau.de/Graphlet/GML/), hence we need the GML translator to translate the extraction result into GML file format. The tasks of node size computation and color assignment are also implemented in the GML translator. The data of the internal class model is the base data for different views, and the structure was described in Section 3.

Graph representation of methods and fields (Fig.3)

In MFV-Class, there are three kinds of node: method node, field node and reference class node. We represent methods using colored boxes of various sizes and shapes, represent fields using colored cycle of various sizes, represent a reference class using black ellipse whose ratio of width and height is 2:1. For the methods box of "This_Class", we use the metric NOC for the height and the NOLV

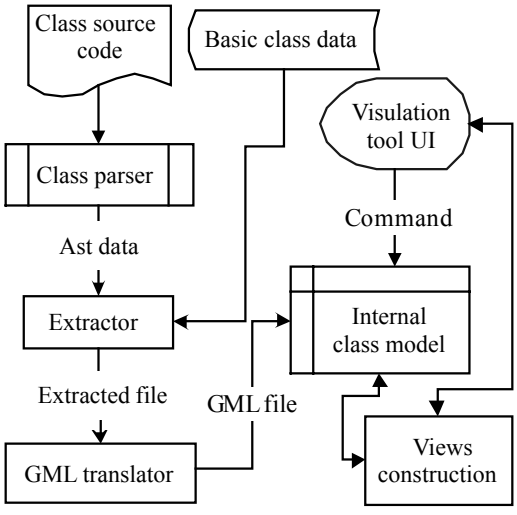


Fig.2 Architecture of MFV-Class

for the width. For the field cycle of "This_Class", we use the metric NOMR for the diameter. For the methods and fields that do not belong to "This_Class", we draw them with their default sizes. The color schemas of relations (edges) and nodes are listed in Table 1 and Table 2, respectively.

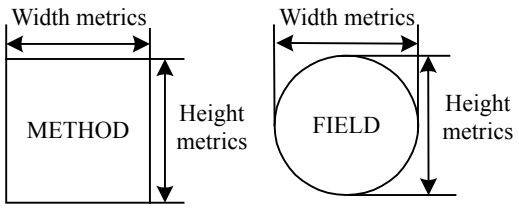


Fig.3 Graphical representation of methods and fields using metrics

Table 1 A color schema for different edges

Relation name	Relation description	Edge color
Call	One method invokes another method	Black
Create	One method invokes constructor of one class	Green
Write	One method writes value to one field	Blue
Read	One method reads value to one field	Orange
Include	One reference class includes its methods and fields	Red

Table 2 A color schema for different nodes*

Node type	Color
Reference_Class	Gray
Protected methods or fields of This_Class	Green
Private methods or fields of This_Class	Blue
Public methods or fields of This_Class	Red
Methods or fields of Super_Class	Yellow
Package accessed methods or fields of This_Class	Cyan
Methods or fields of Reference_Class	Black

*The underlined letters of word are used for showing the node color schema when those figures are printed in a white-and-black printer

CASE STUDY

As an example, MFV-Class is used as an aid to understanding the VGJ project, whose basic data are shown in Table 3. Fig.4 to Fig.10 are the example views created from the comprehension process. For representation clarity, we have omitted the prefix "EDU.auburn.VGJ" of some class names in these figures. These views can be classified into three kinds: class internal views such as Fig.8 (data flow views) and Fig.6 (methods invocation views); class external views such as Fig.7 (public member view) and Fig.4 (reference member view); inheritance view such as Fig.5.

Table 3 Basic statistical data of VGJ

System name	VGJ
Lines of code	>15 KLOC
Number of packages	10
Number of classes	46
Number of methods	527
Number of fields	457

To ease the task of exploring these views, MFV-Class provides the function to show node name at different levels of detail. For method node name, there are four options: name only, with parameters, with modifiers, and no name. For field node name, the options are: name only, with field type, and with modifiers. Fig.9 shows a full view without the name of EDU.auburn.VGJ.graph.Set, which provides a general flavor of the class. Fig.10 is a full view with the full name of EDU.auburn.VGJ.graph.Set, which gives more detailed contents of the class.

In the views of MFV-Class, the node area represents the proportion that a particular node constitutes in the class. For the full view of a class, we define the ratio of total area of method nodes to total area of field nodes as R_{M2F}. We can group the classes in the OO system by the value of R_{M2F}. Fig. 11 shows the distribution of $\ln R_{M2F}$ in the VGJ project. In this figure, the prefix "EDU.auburn.VGJ." is omitted from all the class names listed at right part; instead, only classes' ordinal

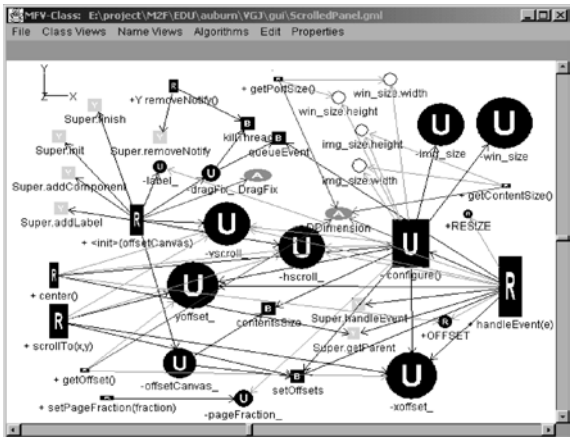


Fig.4 Reference member view of gui. ScrolledPanel

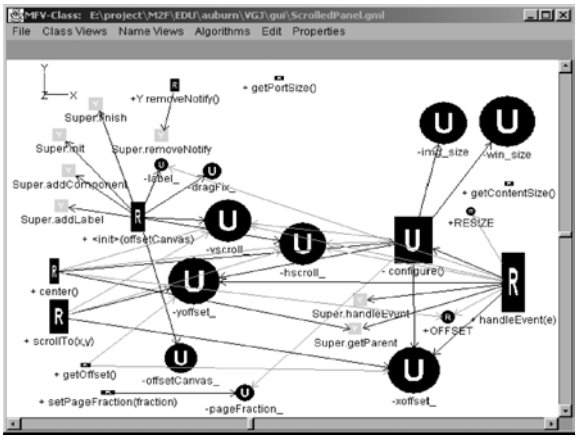


Fig.5 Inheritance view of gui. ScrolledPanel

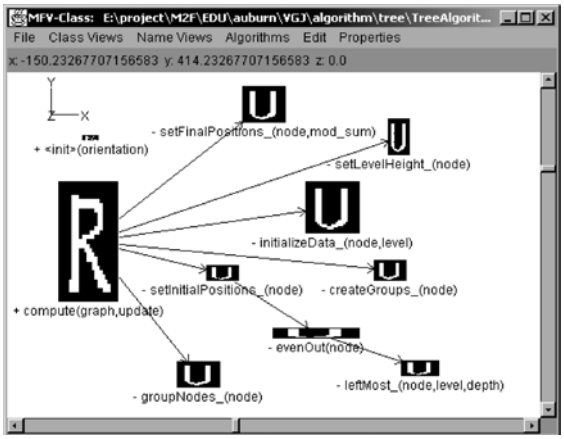


Fig.6 Control flow view of algorithm.tree.TreeAlgorithm

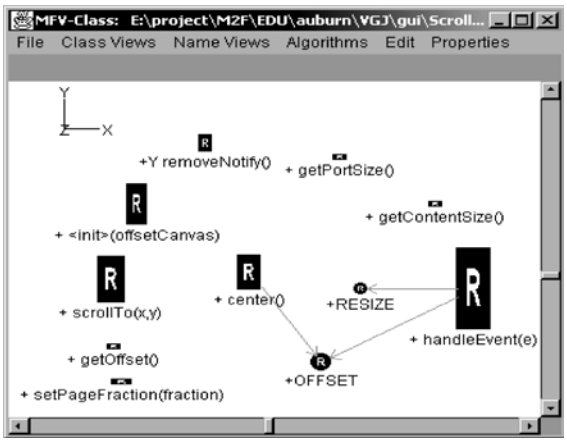


Fig.7 Public member view of gui. ScrolledPanel

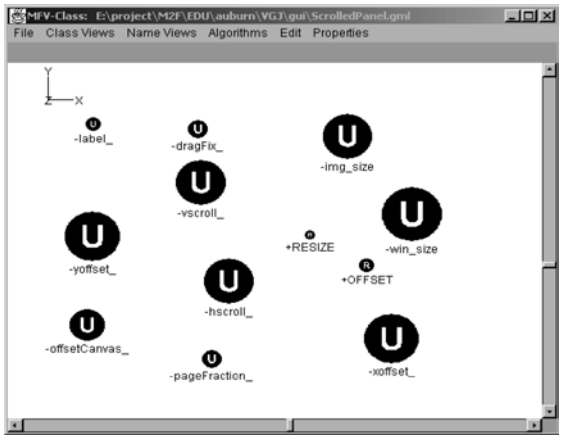


Fig.8 Data flow view of gui.ScrolledPanel

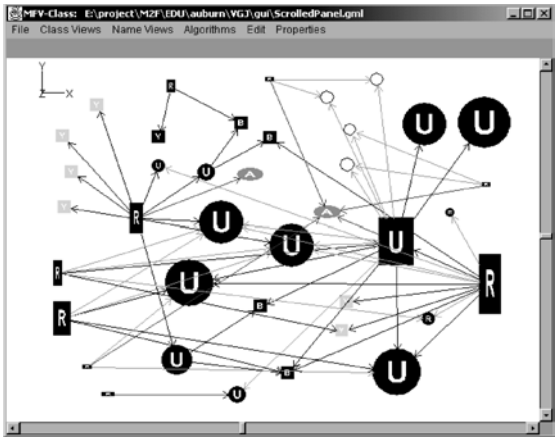


Fig.9 Full view without name of gui.ScrolledPanel

numbers are used in the distribution graph. Note that the figure does not include the 2 interfaces and 4 classes which have no field node in the VGJ project.

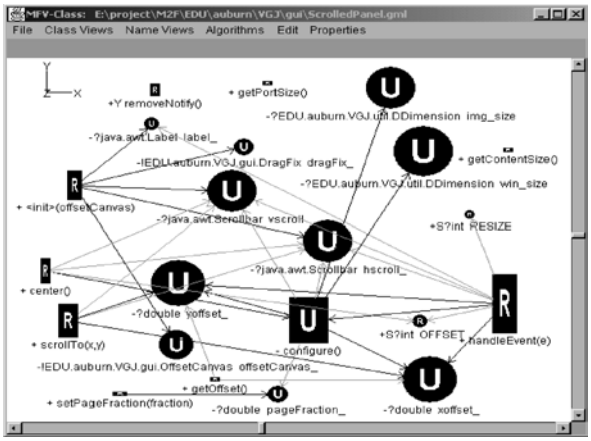


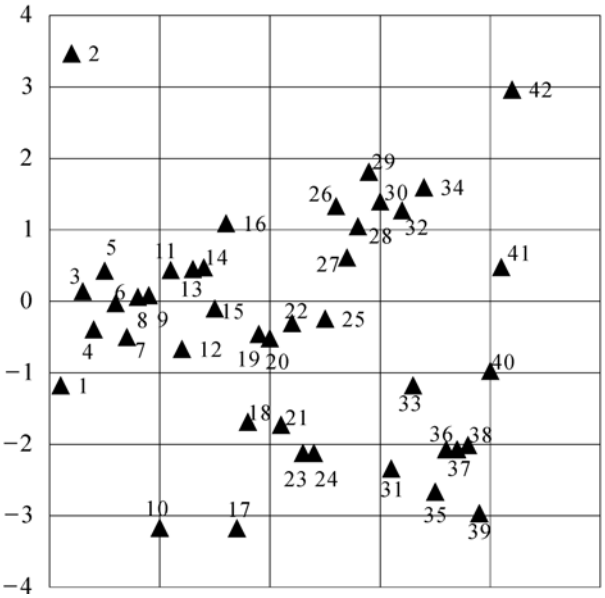
Fig.10 Self member view with full name of gui.ScrolledPanel

Based on the $\ln R_{M2F}$ value of each class, classes in the VGJ project can be classified into three types as listed in Table 4.

In Table 4, it is more difficult and important to understand the last type classes than the first two types, because they are mapped to the domain model and have more complex logic and structure than the others. To understand each of them, the multiple views are more helpful to us. The resultant types of classes (i.e. operation type, data type and domain type classes) also confirm that our class comprehension model can indeed help actual class structure understanding.

CONCLUSION

Classes are key software components in an object-oriented software system. In many industrial



1	algorithm.cartegw.BicornectData	22	graph.NodePropertiesDialog
2	algorithm.cartegw.BicornectGraph	23	graph.Set
3	algorithm.cgd.CGDAlgorithm	24	gui.DragFix
4	algorithm.cgd.Clan	25	gui.GraphCanvas
5	algorithm.cgd.ClanTree	26	gui.GraphEdit
6	algorithm.cgd.Partition	27	gui.GroupWindow
7	algorithm.shawn.Queue	28	gui.GroupControl
8	algorithm.shawn.Spring	29	gui.GroupWamingDialog
9	algorithm.tree.TreeAlgorithm	30	gui.InputDialog
10	algorithm.tree.TreeAlgorithmData	31	gui.Panel
11	graph.AlgPropDialog	32	gui.PSdialog
12	graph.AngleControl	33	gui.ScrolledPanel
13	graph.AngleControlPanel	34	gui.TextOutDialog
14	graph.Edge	35	gui.ViewportScroller
15	graph.EdgePropertiesDialog	36	util.DDimension
16	graph.FontPropDialog	37	util.DDimension3
17	graph.GMILlexer	38	util.DPoint
18	graph.GMILobject	39	util.DPoint3
19	graph.Graph	40	util.DRect
20	graph.Node	41	util.Matrix44
21	graph.NodeList	42	VGJ

Fig.11 The distribution of $\ln R_{M2F}$ in VGJ project

Table 4 Class type in VGJ project

Type	Definition	View character description	Examples
Operation type	$\ln R_{M2F} > 0.5$	Have large number of call edges, and few read or write edges	Most of the classes in EDU.auburn.VGJ.gui
Data type	$\ln R_{M2F} < -0.5$	Nearly no call edges; used for data structure definition	Most of the classes in EDU.auburn.VGJ.util
Domain type	$-0.5 < \ln R_{M2F} < 0.5$	Have average number of call edges and field reference edges	Most of the classes in EDU.auburn.VGJ.graph and EDU.auburn.VGJ.algorithm

OO software systems, there are classes that have complicated structure and relationships. Providing a special CASE tool to aid single class comprehension is absolutely necessary for object-oriented software comprehension and reverse engineering. We have developed a software visualization tool (viz, MFV-Class) which aids a user to understand a class by providing a variety of views. The tool can show a very large OO system by different graph attributes, such as size, shape and color, with no other recognition burden to be imposed. The tool also visualizes three class metrics to enable the user to be on focus during the exploration. The results from our case study validate the proposed comprehension model and the usability of the MFV-Class tool.

The views provided by MFV-Class are at the same abstract level of information as that of the class source code. The tool provides an easy-to-understand representation of source code and partly supports the comprehension process, but it does not provide a semantic description at a higher abstract level. How to get the semantic descriptions for a class is one of the future directions for our OO software reverse engineering research.

References

Brooks, R., 1983. Toward a theory of comprehension of computer programs. *International Journal of Man-Machine Studies*, **18**(6):542-554.

DeBaud, J.M., Moopen, B., Rugaber, S., 1994. Domain Analysis and Reverse Engineering. Proceedings of the 1994 International Conferences on Software Maintenance. IEEE Computer Society Press, Victoria, Canada, p.326-335.

Derry, S., 1996. Cognitive schema theory in the constructivist debate. *Educational Psychologist*, **31**(3/4):163-174.

Li, B.X., Liang, J., Zhang, Y.X., Fan, X.C., Zheng, G.L., 2000. A framework for analyzing Object-oriented programs based on class hierarchy graph. *Journal of Software*, **11**(5):694-700 (in Chinese).

Littman, D.C., Pinto, J., Letovsky, S., Soloway, E., 1987. Mental models and software maintenance. *Journal of Systems and Software*, **7**(4):341-355.

Michele, L., Stephane, D., 2001. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2001), ACM Press, New York, p.300-311.

Pennington, N., 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, **19**(3):295-341.

Zhang, Z.M., Zhuang, Y.T., Pan, Y.H., 2003. Object-oriented software reverse engineering. *Journal of Computer Research and Development*, **40**(6):899-906.