A CASE Tool Platform for Object-Oriented Program

– **Japid**: Its Design and Implementation –

Yoshinari HACHISU

Ph. D Dissertation

A CASE Tool Platform for Object-Oriented Program

– **Japid**: Its Design and Implementation –

School of Engineering,
Nagoya University

Yoshinari HACHISU

Abstract

Object-oriented programming has become very popular. The object-oriented features such as inheritance, encapsulation, and dynamic method invocation are very powerful and useful for software development. To support the lower phases of software development such as implementation and maintenance, many techniques and CASE (Computer Aided Software Engineering) tools such as a dependence analyzer for traditional procedural programs have been proposed and implemented. However, many of them are not applicable to object-oriented programs without modification because of the above object-oriented features.

Software engineers who propose this modification or new techniques for object-oriented programs have to verify their techniques by applying them to programs actually. They sometimes need to make routines that analyze source programs syntactically and semantically like as a compiler does. CASE tool developers also need to make these routines for designing and implementing CASE tools. However, making analysis routines is very laborious work. If there are a generic analyzer and a repository that stores enough information of analyzed programs, each CASE tool developer can avoid such laborious work. We call an environment like this a **CASE tool platform**.

In this thesis, we propose **Japid**, a new CASE tool platform for an object-oriented language, Java. We establish three research goals with Japid: (1) letting developers free from laborious work such as making a syntax analyzer, (2) providing basic techniques for software development, and (3) developing new techniques for software development.

First, to achieve the research goal (1), we show the design and implementation of Japid. Japid is designed to meet following requirements: (i) providing enough information of source programs for developing CASE tools, (ii) providing a method for restructuring information of analyzed programs, and (iii) supporting changing source programs. The first requirement is related to the software model: what kind of abstract syntax tree is stored to a repository. The software model that we adopt is designed at the fine grain level; we also obtain coarse grained information by selecting and composing

fine grained information provided by a platform. The second requirement is about view definition. Japid provides the view definition mechanism. Because a view is defined as classes of Java, a developer can reuse views defined by other developers. The last requirement is important because some CASE tools are developed for changing source programs: for example, a tool that removes unused variables and dead codes. Japid supports changing source programs under some constraints such as syntactic constraints, which guarantee changed programs to be free from syntax errors. This is implemented using the view definition mechanism. We show the effectiveness of Japid by some experiments: Japid provides enough information, API (Application Programming Interface), and acceptable performance for developing CASE tools.

Second, to achieve the research goals (2) and (3), we propose a new control and data dependence graph, *OSDG* (Object-Oriented System Dependence Graph). Dependence analysis is a basic, yet important technique for software development. The OSDG is built at the *expression-grain* level. It can express method composition and a data dependence in a statement that has multiple occurrences of the same variable, such as `a = f(a, g(a))`. We also propose a method for decreasing the amount of nodes and edges by changing granularity of a graph: we translate a graph that consists of *expression* nodes into a graph that consists of *statement* nodes or of *method* nodes. Owing to this method, we can express our concerning point with a precise graph at a fine grain level and other points with brief graphs at a coarse grain level. We show implementation of the OSDG using Japid. Japid is useful for constructing an OSDG because it provides fine grained information such as statements and expressions. We define nodes and edges of the OSDG as a user defined view of Japid. We also implement a method for changing granularity of a dependence graph by using Japid.

Finally, to achieve the research goal (3), we propose three approaches to specialization of object-oriented programs. The first approach is *class fusion* and *reduction*, which are techniques merging closely associated classes into a single class. The second approach is *class slimming*, which removes unused methods and variables from classes of a class library or from classes enlarged by class fusion and reduction. The last approach is *static analysis of dynamic invocation*. It replaces dynamically-dispatched method invocation with statically-dispatched one. These approaches resolve a dilemma of whether to write a program elegantly for easy maintenance or to tune it up for good performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

1.1.1 CASE Tool Platform

Object-oriented programming has become very popular. The object-oriented features such as inheritance, encapsulation, and dynamic method invocation are very powerful and useful for software development. However, it takes long time that some developers who have developed in other programming paradigms such as procedural programming and functional programming become familiar with the object-oriented paradigm and bring out its power. Using CASE (Computer Aided Software Engineering) tools that support development of object-oriented software, they will get accustomed to its paradigm in short time.

CASE tools are also very useful for rapid and easy development for other developers who have gained experience in the object-oriented paradigm. Especially, in the lower phases of software development such as implementation and maintenance, developers sometimes need to analyze and understand programs. Many techniques such as dependence analysis for traditional procedural programs have been proposed and implemented to support these activities. However, many of them are not applicable to object-oriented programs without modification because of the above features of object-oriented programs.

Software engineers who propose this modification or a new technique for object-oriented programs have to verify their ideas by applying them to programs actually. They sometimes need to make routines that analyze source programs syntactically and semantically like as a compiler does. CASE tool developers, who sometimes develop tools using techniques proposed by software engineers, also need to make these routines for designing and imple-

menting CASE tools. However, making them is very laborious work. If there are a generic analyzer and a repository that stores enough information of analyzed programs, each CASE tool developer does not need to make analysis routines by himself (Figure 1.1). We call an environment like this a **CASE tool platform**. In this chapter, we often call it a *platform*, simply.

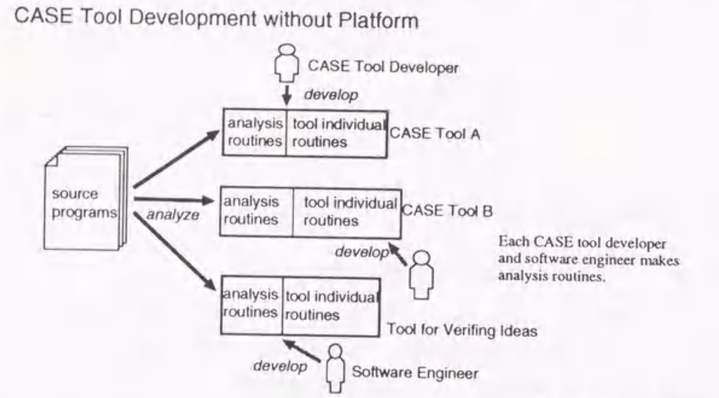


Figure 1.1: CASE Tool Development

Requirements of CASE Tool Platform

A CASE tool platform for supporting lower phases of software development should satisfy following three requirements.

Req. 1. A platform should provide enough information of source programs for developing CASE tools.

Req. 2. A platform should provide a method for restructuring information of analyzed programs.

Req. 3. A platform should support changing source programs.

The first requirement is related to a software model. In general, a platform stores and provides an abstract syntax tree or its variant. A software model is a model for making an abstract syntax tree [1].

Information provided by a platform does not always match information needed by a CASE tool, because each CASE tool has different view points of programs. For example, one focuses on associations between classes, while another focuses on data dependences between statements. A platform should adopt a software model that can cover many view points of programs. Here, we discuss a software model from a view point of granularity of information: that is, what kinds of symbol nodes should compose an abstract syntax tree.

When provided information is coarser than information needed by a CASE tool (i.e., a provided syntax tree is more abstract than a necessary tree), CASE tool developers must make analysis routines for getting finer information. For example, we suppose that a platform provides only information about classes and methods, i.e., a provided tree consists of *class* and *method* nodes. When a developer[2] would like to make a data dependence analyzer, he or she must make routines that *re-parse* source programs for getting information about statements. In this case, such an environment is not called a CASE tool platform any longer because developers must make analysis routines by theirselves.

When provided information is finer (i.e., a necessary tree is more abstract than a provided syntax tree), CASE tools can obtain desired information by selecting and composing some provided information. For example, we suppose that a platform provides information about classes, methods, and statements. When a CASE tool needs information about method overriding, this tool can get desired information by selecting information about classes and methods. From this discussion, a *fine grained* software model is suitable for a CASE tool platform.

The second requirement is about view definition. We have already described that CASE tools have various view points and that they can obtain desired information by restructuring fine grained information. An idea for restructuring is analogous to view definition of database systems: while the software model provided by a platform is considered as a conceptual schema

[1] We show more precise description of an abstract syntax tree, its variant, and a software model in Chapter 2.

[2] From this paragraph, a *developer* means a CASE tool developers.

of the database system, a model restructured by a developer is considered as an external schema[3][8].

View definition supported by a platform brings two advantages. The first advantage is easy restructuring. With support of a platform, developers can restructure information easily. The second advantage is data sharing. When each CASE tool restructures information on its own way, it is difficult for a CASE tool to reuse information restructured by another CASE tool. If a platform provides a framework for restructuring, some CASE tools can share restructured information easily. This sharing is called *data integration* of CASE tools[5].

The last requirement is very important because some CASE tools are developed for changing source programs. Some kinds of changes of programs are monotonous work: for example, changing the name of a variable. CASE tools should support these kinds of changes. There are some common constraints on changes of programs: for example, changed programs can be compiled without errors. It is wasteful for each CASE tool developer to make routines that change programs under common constraints. A platform should provide a method for changing programs.

1.1.2 Related Work

In this section, referring to three requirements of a CASE tool platform, we show related work.

PCTE

PCTE (Portable Common Tool Environment) is an international standard for an open repository[36]. A key feature of it is the object management system. It is designed for data integration and it has the view mechanism: each tool defines its own view by aggregating existing schemas. It also has ability to define access rights and locks on an object, and it manages not only source programs but also specification documents. However, granularity of data managed by PCTE is coarse (e.g. a file and module) and its object management system does not support changing source programs.

SoftBench

C/C++ SoftBench is an integrated set of programming tools and a tool integration platform[19]. It supports control and display integration of CASE

[3]Implementation of a repository (e.g., a simple file system, relational database management system, etc.) is considered as an internal schema.

tools: a tool communication architecture is designed, and an OSF/Motif style graphical user interface is provided. They also provide an integrated software development environment targeted at the program construction, test and maintenance phases of software development. The SoftBench product is composed of some components: program editor, C++ class graph editor, debugger, static analyzer, and so on.

Static analyzer is an important part for constructing CASE tools. It provides cross-reference information about identifiers: for example, a declared variable and reference to it, relationships of function calls, and so on. But it does not provide information about statements and expressions.

C++ class graph editor supports a change of a class structure. However, another CASE tools cannot reuse this function, and other changes such as changing a variable's name are not supported.

CIA++

The C++ Information Abstractor, CIA++, builds a database of information extracted from C++ programs for the rapid development of C++ programming tools[18]. It manages analyzed data in a relational database and its conceptual schema consists of five entities (*file, macro, type, variable*, and *function*) and four major relationships between them (*include, inheritance, friendship*, and *reference* relationships). It has its own declarative query language, which supports view definition.

Granularity of information provided by CIA++ is coarse: it cannot express statements and expressions. It does not support changing source programs.

GENOA/GENII

GENII is a parser generator. A generated parser translates source programs into a language-independent syntax tree. GENOA is a generator for a program analysis tool. A generated analysis tool reads a language-independent syntax tree, and it produces some output for analysis: for example, data dependence graphs[10]. A GENOA/GENII system bound for the C++ language is implemented and some tools such as a control dependence graph generator are made with it[11].

We consider that GENII and a generated parser provide an abstract syntax tree and that GENOA and a generated analysis tool produce a view of an abstract syntax tree. However, it does not support changing source programs.

REFINE

REFINE is a family of interactive, extensible workbenches for analyzing and reengineering code in programming languages such as C, Ada and so on[29]. REFINE analyzes programs syntactically, and provides analyzed data as an object-oriented database. Operations on REFINE are written in its own Lisp-like language, which has various data types such as a tree and a set.

REFINE provides fine grained data. However, it does not support view mechanism and changing source programs.

Sapid

Our research group has already delivered a CASE tool platform for the C language, **Sapid**. Some useful tools such as a data dependence analyzer[27] and an interpreter[22] are implemented with Sapid[16][31].

Sapid models the syntax and semantic of the C language as 12 entities (*file, identifier, expression, operator*, etc.) and 29 relationships. It provides some high level API functions, such as a data and control dependence and interpreter functions.

Sapid provides fine grained information. The model and the repository of Sapid are designed to support changing source programs, and a mechanism for it has been already proposed[15]. However, it has not been implemented yet.

Summary

We show satisfaction of three requirements in each platform in Table 1.1. No platform satisfies all three requirements.

Table 1.1: Satisfaction of Requirements in Each Platforms

Platform	Req. 1 data grain	Req. 2 view def.	Req. 3 prog. change	Target Lang.
PCTE	coarse	○	×	C, Ada
SoftBench	identifier	×	△	C, C++
CIA++	coarse	○	×	C++
GENOA/GENII	fine	○	×	C++
REFINE	fine	×	×	C, Ada
Sapid	fine	×	△	C

1.2 Research Goals

In this thesis, we propose a new CASE tool platform, **Japid**, for the object-oriented language, Java. We have applied some techniques of Sapid to Japid. We have designed Japid to satisfy three requirements:

1. Japid provides fine grained information of analyzed programs. Its software model consists of 15 entities (*file, class, routine, variable, statement, expression, operator, literal*, etc.) and 51 associations (*class_routine, routine_body, statement_expression, expression_operator, refer_literal*, etc.). It provides the software model as classes of Java and an abstract syntax tree as objects of Java.

2. It permits CASE tool developers to define their own views. This mechanism is performed in two steps. First, a developer defines his or her software model as classes of Java. Second, Japid produces an abstract syntax tree from his or her classes.

3. It supports changing source programs under some constraints such as syntactic constraints, which guarantees changed programs to be free from syntax errors. This is implemented using the view definition mechanism.

In this thesis, we establish the following three research goals using Japid:

(1) letting developers free from laborious work such as making a syntax analyzer

(2) providing basic techniques for software development

(3) developing new techniques for software development

The first goal is an essential part of a CASE tool platform. We have already discussed this goal.

The second goal means that Japid becomes a *platform* for applying and sharing basic techniques for software development. Some techniques such as dependence analysis are used by many CASE tools. If someone implements these techniques with Japid, other developers can reuse them.

The last goal means that Japid becomes a *platform* for developing new techniques. If a platform provides enough information of programs and basic techniques, new ideas can be verified using them.

1.3 Object-Oriented Language, Java

Many techniques have been proposed and verified to simple programming languages such as a subset of Pascal because practical programming languages such as C and C++ are very complex. Complexity causes two problems. First, complexity makes it difficult to make analysis routines for checking new ideas. A platform is a solution of this problem. Second, complexity of a language makes an algorithm of a new technique complex. We solve this problem by selecting Java, which is a simple, yet powerful object-oriented language. In this section, we briefly explain Java's features.

Java is a programming language with some good features: for example, simple, object-oriented, architecture neutral, and so on[32][35]. It has been recognized as one of the practical general purpose languages like as C and C++, and general applications such as a word processor are written in Java.

Java is very similar to C++, but it is much simpler. The object-oriented facilities of Java are essentially those of C++. But the unnecessary complexities of C++ such as pre-processor, multiple inheritance, operator overloading, pointer arithmetic, etc., are removed in Java. Java supports automatic garbage collection. It makes Java programs a lot simpler to write because programmers do not have to worry about memory management.

A Java compiler translates source programs into *byte-code* instructions that have nothing to do with a particular computer architecture. Byte-codes can run on any platforms that support Java. It is not necessary to recompile Java programs to run on a new machine.

1.4 Overview of This Thesis

In Chapter 2, to achieve the research goal (1), we show the design and implementation of our CASE tool platform, Japid. It is designed to meet three requirements for a CASE tool platform. We carried out some experiments and developed some tools to demonstrate effectiveness of Japid: it provides enough information of source programs and API (Application Programming Interface) for developing CASE tools.

In Chapter 3, to achieve the research goals (2) and (3), we propose a new dependence graph, *Object-Oriented System Dependence Graph* (OSDG). Dependence analysis is a basic, yet important technique for software development and is used for program optimization, understanding, etc. We show advantages and disadvantages of analysis methods proposed previously. The OSDG is designed to overcome these disadvantages.

The OSDG is built at the *expression-grain* level for providing more precise information than traditional graphs. The OSDG enables representing data dependences of method composition and of a statement that has multiple occurrences of the same variable, such as `a = f(a, g(a))`. Generally, fine grained analysis produces a large amount of data. We also propose a method for decreasing the amount of nodes and edges by changing granularity of a graph [4]. Owing to it, we can express our concerning point with a precise graph at a fine grain level and other points with brief graphs at a coarse grain level. We developed tools that construct an OSDG using Japid and show their performance. Japid is useful for constructing an OSDG because it provides fine grained information such as statements and expressions. We define nodes and edges of the OSDG as a user defined view of Japid. We also implemented changing granularity of a dependence graph and show that a coarse grained graph is constructed in shorter time and less memory than a fine grained graph.

In Chapter 4, to achieve the research goal (3), we propose new approaches to specialization of object-oriented programs. We classify specialization of object-oriented programs into two kinds: *intra-class* specialization and *inter-class* specialization. The former specializes statements, expressions, and method invocations *in a class*: for example, loop unfolding and method inlining. Specialization techniques for procedural programs can be applied as intra-class specialization. The second type, inter-class specialization is based on the structure *among some classes*. This specialization is peculiar to object-oriented programs. In this thesis, we propose three approaches to inter-class specialization.

First, we propose *class fusion* and *reduction*, which are techniques for merging closely associated classes into a single class. Using them, we can apply intra-class specialization to more classes. The second approach is *class slimming*, which removes unused methods and variables from classes of class libraries or from classes enlarged by class fusion and reduction. The last approach is *static analysis of dynamic invocation*, which replaces dynamically-dispatched method invocation with statically-dispatched one. These specialization techniques resolve a dilemma of whether to write a program elegantly for easy maintenance or to tune it up for good performance. We carried out some experiments that demonstrate effectiveness of our approaches. We designed and implemented some tools for experiments using Japid and the

[4] *Granularity* of a dependence graph is represented as kinds of its symbol nodes. For example, at fine grain level, a graph consists of *expression*, *statement*, *method*, *variable*, and *class* symbol nodes. At coarse grain level, a graph consists of *method*, *variable*, and *class* symbol nodes.

OSDG.

In Chapter 5, as conclusion, we show achievements of our three research goals and future work of our research.

Chapter 2

Japid: A CASE Tool Platform for Java

To achieve the research goal (1), we propose a CASE tool platform, Japid, in this chapter. First, we explain a software model, which is a model for making an abstract syntax tree. Second, we show the design and implementation of Japid. Japid satisfies three requirements for a CASE tool platform.

1. It provides fine grained information of programs.
2. It permits users[1] to define their own views.
3. It supports changing source programs under syntactic and semantic constraints. It is implemented using user defined views.

Finally, we demonstrate its effectiveness by some experiments and some CASE tools.

2.1 Software Model

We can translate a source program into the tree structure, called the *concrete syntax tree*, according to the grammar. Nodes of a concrete syntax tree are *terminal* or *non-terminal symbols*[2]. Because a concrete syntax tree has all symbols of a source program, it is apt to become very large. We often eliminate redundant nodes from it and make a simpler tree. This tree is called an *abstract syntax tree*[1]. Various abstract syntax trees are constructed according to the purposes of CASE tools. We call a model to make an abstract syntax tree a **software model**.

[1] In this chapter, *users* mean users of Japid and CASE tool developers.
[2] A *terminal symbol* is often called a *token*.

2.1.1 J-model

In our CASE tool platform, source programs are managed at the fine grained level. Our abstract syntax tree is close to a concrete syntax tree. Users can define their own views by selecting and composing classes and associations that Japid provides. In this section, we explain our software model for Java, called **J-model**. It is written in Object Modeling Technique notation[30], and consists of 15 classes [3] and 51 associations. Classes of J-model represent symbols of the Java's grammar: for example, the *Variable, Statement, Expression*, and so on. Associations of J-model are classified into two kinds: *syntactic* associations and *reference* associations. We show the class diagram of J-model in Figure 2.1 and an instance diagram of it in Figure 2.2 [4].

In this section, we explain some important classes and associations of J-model.

File

In J-model, an instance of the *File* class is the root node of an abstract syntax tree. In Java, a source program, which consists of classes and interfaces, is written in a file, and a compiler compiles its file. The *File* class shows a source program file and it also shows the start symbol of the grammar, $\langle CompilationUnit \rangle$.

Identifier

An identifier is a name defined by programmers to identify an element of program such as a class, a variable, and a method. We model identifiers of Java as four classes such as the *Class*, *Interface*, *Variable*, and *Routine* classes. They represent each declared identifier and declaration of it. For example, when method declaration is defined as follows[5], the *Routine* class [6] represents $\langle Identifier \rangle$ and $\langle MethodDeclaration \rangle$.

$$\begin{aligned} \langle MethodDeclaration \rangle \quad :== \quad & [\ \langle Modifiers \rangle\ ]\ \langle Type \rangle \\ & \langle Identifier \rangle\ \text{“(”}\ \langle Parameters \rangle\ \text{“)”} \\ & [\ \texttt{“throws”}\ \langle Types \rangle\ ] \\ & [\ \langle MethodBody \rangle\ ] \end{aligned}$$

[3] In this thesis, the word class has two meanings, class of J-model and class of Java. To clear this difference, we print class of J-model in italic.

[4] An instance diagram of a software model is an abstract syntax tree. One of J-model becomes a graph because of reference associations (see later).

[5] It is written in extended Backus-Naur Form. “[..]” surrounding shows optional items.

[6] The *Routine* class shows a method and a constructor (see later).

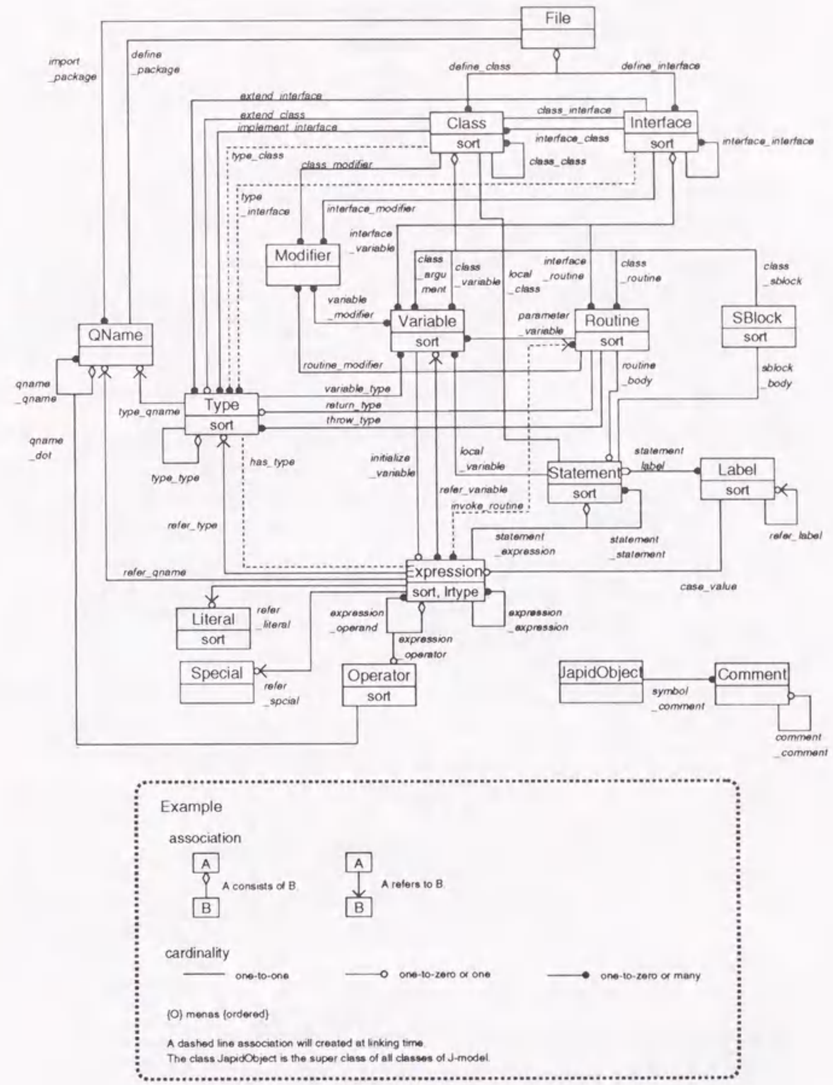
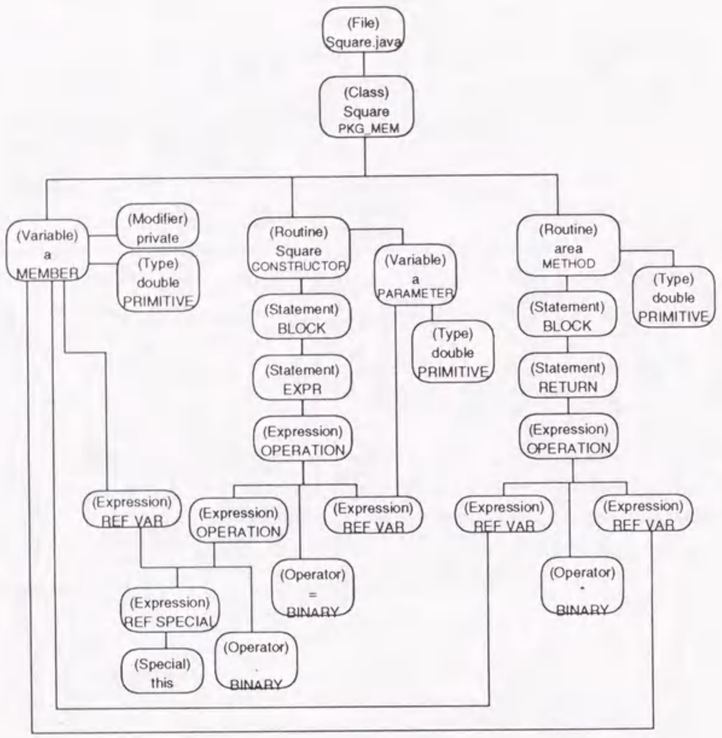


Figure 2.1: Class Diagram of J-model




```
class Square {
    private double a;

    Square(double a) {
        this.a = a;
    }

    double area() {
       return a*a;
    }
}
```

Square.java

Figure 2.2: Instance Diagram of J-model

The *Class* class shows a class of Java and the *Interface* class shows an interface of Java. We classify classes of Java into five kinds: package member classes, nested top-level classes, member classes, local classes, and anonymous classes [7]. I also classify interfaces in the same way. The attribute *sort* shows this classification.

The *Variable* class shows a variable. In Java, variables are classified to three kinds: member variables, parameters of a method, and local variables. The *Variable* class has the attribute *sort* to show this classification.

The *Routine* class shows a method or a constructor. The attribute *sort* shows this distinction. Because in Java, programmers can define methods that have the same name and different parameters, we use the pair of the name and the parameters of a method to distinguish them. This pair is called the *signature* of a method.

Control and Operation

We consider the program execution as controlling flows of data and operating data. In the grammar, a control is shown as $\langle Statement \rangle$ and an operation is shown as $\langle Expression \rangle$. We model each of them as the *Statement* and the *Expression* class.

We classify statements into some kinds: sequence, selection, repetition, and so on. We also classify expressions into some kinds: reference to a variable or a literal, operation with an operator, method invocation, and so on. These classifications are shown by the *sort* attributes. The *Expression* class also has the attribute *lrtype*. It shows whether an expression is a left-hand side expression or a right-hand side expression.

We show instance diagrams of the *Statement* class in Figures 2.3, 2.4, and 2.5.

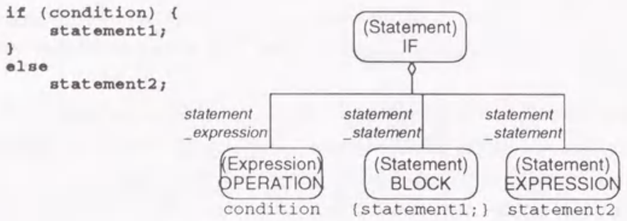
Syntactic Association

A syntactic association is an association between a left-hand side and a right-hand side of the grammar. For example, in the following grammar

$$\langle Statement \rangle ::= \langle Expression \rangle \text{ ";"}$$

there is a syntactic association between the *Statement* class and the *Expression* class. This association is also called a *consist-of* association because we consider that statement consists of the expression.

[7] Member classes, local classes, and anonymous classes are called *inner classes*[2][14] They are added in Java 1.1.



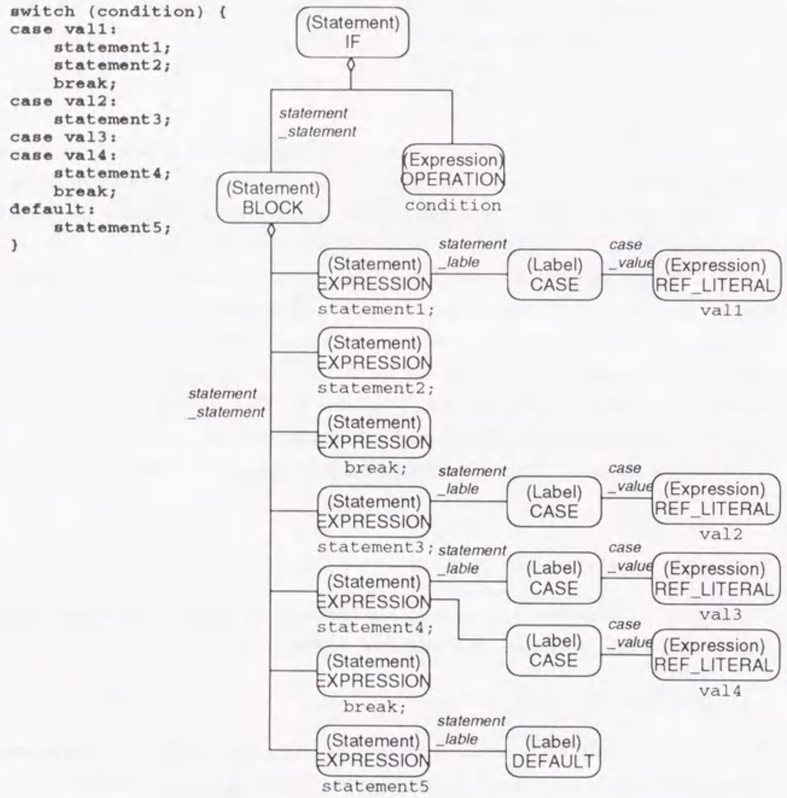
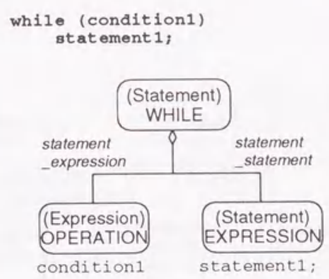
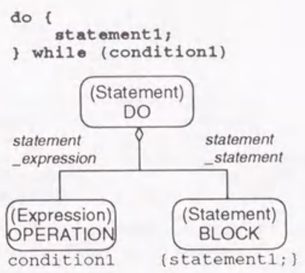
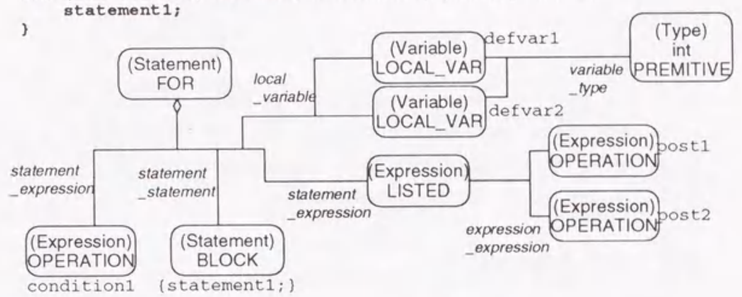


Figure 2.3: Selection Statement







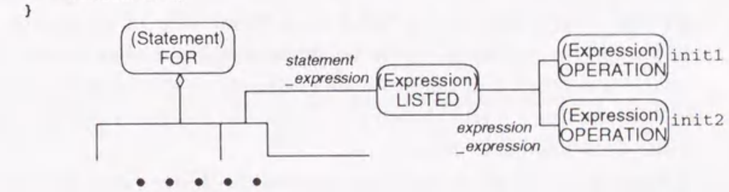


Figure 2.4: Repetition Statement

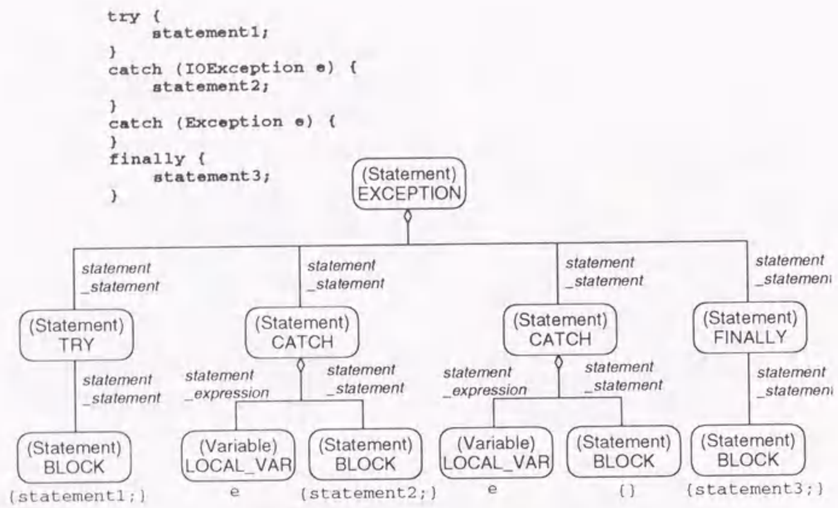


Figure 2.5: Exception Statement

A tree that consists of objects of J-model classes and links of syntactic associations is an abstract syntax tree. We visit all objects in a source program file by traversing objects from the object of the *File* class, which is the root of a tree, along syntactic associations.

Reference Association

A reference association is an association between a declared identifier and reference to it: for example, between the *Variable* class, which represents a declared variable, and the *Expression* class that represents referring to a variable. This association is based on a name table of a compiler. In the program in Figure 2.6, there are reference associations between the member variable `a` in line 2 and the expression referring to the variable `a` on the left-hand side in line 4, and between the parameter `a` in line 3 and the expression on the right-hand side in line 4.

There is a reference association between the *Method* class and the *Expression* class that represents a method invocation. Because a method invocation in Java is bound dynamically at run-time, we classify reference associations between the *Method* class and the *Expression* class into three kinds: the **MUST-be-invoked**, **MAY-be-invoked**, and **is-NOT-invoked** association. A method invocation on an object that is instantiated by the `new`


```
1: class Sample {
2:     private int a;
3:     public Sample(int a) {
4:         this.a = a;
5:     }
6: }
```

Figure 2.6: Sample Program

operator is the MUST-be-invoked association. When an object invokes a *final* method, which must not be overridden by subclasses in Java, it is the MUST-be-invoked association, too. In this case, a method invocation can be bound statically. When an object invokes an *abstract* method, which has no method body, it is the is-NOT-invoked association. In this case, a concrete method of a subclass is invoked. Other method invocations are the MAY-be-invoked associations: for example, an invocation on an object that is a method parameter. In this case, a method is bound dynamically: it or one of methods overriding it is invoked. Owing to this classification, in inter-method analysis, users know whether they must analyze subclasses' overriding methods or not. To perform inter-method analysis more precisely, we have to find MAY-be-invoked invocations that can be resolved to MUST-be-invoked invocations. For example, when a method is not overridden by any subclasses, invocations of this method are MUST-be-invoked. We show some analysis methods for this purpose in Chapter 4.

A reference association is a semantic association and it is not represented by the grammar. Because of a reference association, an instance of J-model is not a tree, but a graph. We call this variant of an abstract syntax tree a semantic graph.

Differences between Models of Sapid and Japid

J-model is modeled referring to I-model, the software model of Sapid (Figure 2.8). However, there are some differences between them:

- In I-model, an identifier and its declaration are shown as the *identifier*, the *declarator*, and the *declaration* class according to the grammar. This strict representation makes confusion for CASE tool developers to understand their program structures. For example, there is no direct association between the *function* class, which shows the name of a function, and the *funcdef* class, which shows parameters and the body

```
 1: interface Printable {
 2:     void print();
 3: }
 4:
 5: class Base implements Printable {
 6:     protected String message = null;
 7:     public void print() {
 8:         System.out.println(message);
 9:     }
10:     public void message(String s) {
11:         message = "Base Class: " + s;
12:         print();
13:     }
14: }
15:
16: class Derived extends Base {
17:     public void message(String s) {
18:         message = "Derived Class: " + s;
19:         print();
20:     }
21:
22: }
23:
24: public class VCall {
25:     static void virtualcall1(Base base) {
26:         base.message("Hello World");
27:           // Base#message(String)
28:           //   MAY be invoked
29:     }
30:     static void virtualcall2(Printable obj) {
31:         obj.print();
32:           // Printable#print() is NOT invoked
33:     }
34:     public static void main(String args[]) {
35:         Derived derived = new Derived();
36:
37:         virtualcall1(derived);
38:         virtualcall2(derived);
39:         derived.message("TEST");
40:           // Derived#message(String)
41:           //  MUST be invoked
42:     }
43: }
```

Figure 2.7: Method Invocations

of a function.

The *Class*, *Interface*, *Variable*, and *Routine* classes of J-model are abstractions of identifiers and their declaration.

- In I-model, a statement is not modeled as the *statement* class. A statement is modeled as the *block* class and the *expression* class. The *block* class is classified into four kinds: a sequence of statements (a basic block), a if-statement (a branch block), a block including other blocks (a compound block), and declaration statements of local variables (a declaration block). Other statements such as a *return-statement* and an *expression-statement* are modeled as the *expression* class. This modeling decision also makes confusion.

  In J-model, a statement is modeled as the *Statement* class.

2.2 Japid

2.2.1 Structure

Japid is a CASE tool platform for Java. Japid consists of four parts: the analyzer, the software database, the access library, and the writer. Figure 2.9 shows the overview of Japid.

Analyzer

The analyzer analyzes a source program according to J-model, and stores analyzed information into the software database. J-model is considered as the conceptual schema for the database[8].

Software Database

The software database (SDB) manages objects generated by the analyzer. Because J-model is written in the OMT notation, an object-oriented database management system (OODBMS) is suitable for managing the SDB[25].

In languages such as Pascal, C, and C++, we get an executable binary program by compiling source programs, and linking compiled objects and some libraries. In Java, a source program is compiled and translated into byte-code files. A byte-code file is interpreted by a Java virtual machine (JVM) and the JVM links it with other byte-code files at run-time.

[8]User defined view classes are considered as the external schema (see Section 2.2.2). The two level database is considered as the internal schema (see later).

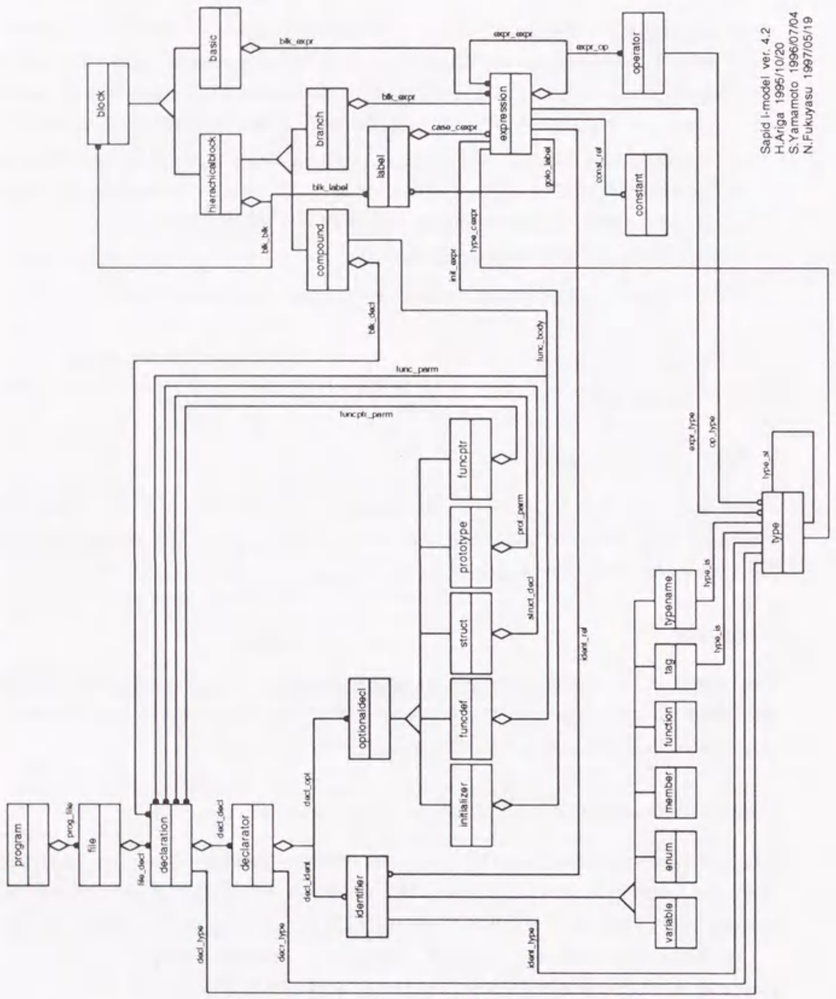


Figure 2.8: Class Diagram of I-model

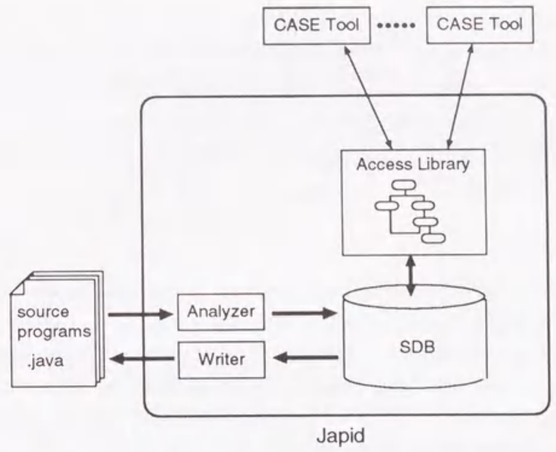


Figure 2.9: Overview of Japid

According to this feature of Java, the SDB is managed at two level in Japid (Figure 2.10). The first level is a fine grained level. A fine grained database manages objects generated by the analyzer, and it is made from every source program. The second level is a coarse grained level. A coarse grained database manages fine grained databases. This is considered as the internal schema in the wide sense [9].

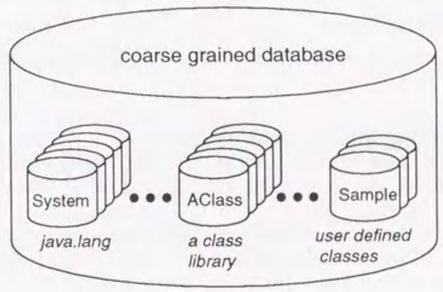


Figure 2.10: Two Level Database

When users want to get information of classes, they get the fine grained databases of these classes from the coarse grained database, and link objects

[9] In the narrow sense, the internal schema of a fine grained database corresponds to the internal schema of Japid. Currently, it is implemented using a simple file system: objects and associations are listed in files.

of them dynamically. In general, because a class of Java uses many classes of class libraries such as a `java.lang` package, linking objects of all used classes statically needs long time and a lot of memories. Dynamic linking solves this problem. It also has good reusability because we can reuse fine grained databases of class libraries, and because we make only databases of updated source programs.

Access Library

The access library, which is written in Java, is a class library for accessing objects in the SDB. Users make CASE tools using it. It represents J-model classes as classes of Java, and translates data retrieved from the SDB into objects in Java. For example, the *Expression* class of J-model is represented as `class Expression {...}` by the access library, and an instance of the `Expression` class represents the data in the SDB. We can consider the SDB and the access library as an OODBMS.

API (Application Programming Interface) methods for getting information from the software database are classified into four kinds: methods for getting classes, getting objects, getting associated objects, and getting an attribute of an object.

A method for getting classes returns J-model classes declared in the access library. Because classes are not the *first class data objects* in Java, we use a string of characters in order to show a class. For example, the *Routine* class is represented as a string `"Routine"`. We get a *true* class by `java.lang.Class#forName(String className)` method [10] and create its object by `java.lang.Class#newInstance()` method in Java.

```
// Extents manages J-model classes
Enumeration classNames = Extents.getClassNames();
   // returns strings "Class", "Routine", "Statement", ...

// Create a Routine class and its instance
java.lang.Class routineClass
    = java.lang.Class.forName("Routine");
Routine routineObj = routineClass.newInstance();
```

A method for getting objects returns a set of objects of a given class. All objects of the access library are managed by `Extents` class. `Extents#getObjects(String className)` returns all objects of a class, `className`: for example, `Extents#getObjects("Routine")` returns all objects of the `Routine` class.

[10] In this thesis, "`#`" separates a class and its method (`ClassName#method()`).


```
Enumeration routines = Extents.getObjects("Routine");
   // returns all objects of the Routine class
```

A method for getting associated objects returns objects associated with a specified object. It is implemented as a method of each J-model class. For example, `Routine#getParameters()` returns the parameters of a method, which are objects of the `Variable` class.

```
Routine r;
...
Enumeration params = r.getParameters();
    // returns parameters of method r
Type retType = r.getReturnType();
    // returns return type of method r
```

A method for getting an attribute of an object returns an attribute of a specified object. It is implemented as a method of each class of J-model. For example, `Variable#getSort()` returns a sort of *Variable* class.

```
Variable var;
...
int sort = var.getSort();
    // returns sort of Variable var
if (sort == Variable.MEMBER) {
    // a member variable
    // Variable.MEMBER is a constant
    //   showing a member variable
    ...
}
else if (sort == Variable.PARAMETER) {
    // a parameter
    ...
}
else if (sort == Variable.LOCAL_VAR) {
    // a local variable
    ...
}
```

We also provide methods for traversing objects along syntactic associations. They are implemented in each J-model class as the method, `traverse(TraverseMethod tm)`. `TraverseMethod` is an interface in Java and declares three methods invoked on traversing. Figure 2.11 shows declaration of `TraverseMethod`. The method `doBeforeTraverse(JapidObject obj)` is invoked before descending to an abstract syntax tree. The

JapidObject class is the superclass of all J-model classes. The method `doAfterTraverse(JapidObject obj)` is invoked after descending to an abstract syntax tree. The method `doMyself(JapidObject obj)` is invoked between `doBeforeTraverse(JapidObject obj)` invocation and `doAfterTraverse(JapidObject obj)` invocation. These are analogous to a pre-order, post-order, and in-order traversal of a tree. We show `traverse` of the `Variable` class in Figure 2.12. When symbol $\langle Variable \rangle$ is defined as follows, method `doMySelf(JapidObject obj)` is invoked at the position of $\langle Identifier \rangle$.

$$\langle Variable \rangle ::== [\langle Modifiers \rangle]\ \langle Type \rangle\ \langle Identifier \rangle\ [\text{``="}\ \langle Expression \rangle]$$

```
public interface TraverseMethod {
    public void doBeforeTraverse(JapidObject obj);
    public void doMyself(JapidObject obj);
    public void doAfterTraverse(JapidObject obj);
}
```

Figure 2.11: TraverseMethod

We show an example using `traverse(Traverse)` in Figure 2.13. The `GetMembers` class is a concrete class of the `TraverseMethod` interface.

Writer

The writer writes back objects in the SDB to source programs. When objects in the SDB are changed, the writer changes source programs. For example, when users want to change a variable's name, they can change a name attribute of an object of the `Variable` class. Then, the writer changes a variable's name in source programs according to modified data in the SDB. This is performed by program change mechanism (see Section 2.2.3).

2.2.2 User Defined Views

We have already described that a CASE tool platform must provide various views of programs. For example, someone who focuses on a class hierarchy will not pay attention to statements and expressions, while another who focuses on control flows may want to specialize the *Statement* class, for example, the *Selection* class and the *Repetition* class. Our access library supports the view definition mechanism.


```
/**
 *  Variable#traverse()
 *
 *  Variable :== [Modifiers] Type Identifier [ "=" Expression ]
 *
 */
public void traverse(TraverseMethod tm) {
    // invoke doBeforeTraverse before traversing a syntax tree
    tm.doBeforeTraverse(this);

    /** start traversing **/
    // modifiers of this variable
    for (Enumeration mods = getModifiers();
         mods.hasMoreElements();) {
        Modifier mod = (Modifier) mods.nextElement();
        mod.traverse(tm);
    }

    // type of this vairable
    type.traverse(tm);

    // variable itself
    tm.doMyself(this);

    // initializer expression
    if (initializer != null) {
        initializer.traverse(tm);
    }

    // invoke after traversing
    tm.doAfterTraverse(this);
}
```

Figure 2.12: `traverse(TraverseMethod)` of *Variable* class

```
/**
 * Gets member variables referred to.
 */
class GetMembers implements TraverseMethod {
    Vector  used = new Vector();    // used variables
    Vector  defined = new Vector(); // defined variables

    /** Do nothing */
    public void doBeforeTraverse(JapidObject obj) {
    }

    /**
     * If parameter obj is use of a member variable,
     *   it is added to used.
     * If parameter obj is definition of a member variable,
     *   it is added to defined.
     */
    public void doMyself(JapidObject obj) {
        if (obj instanceof Expression) {
            Expression expr = (Expression) obj;
            if (expr.getSort() == Expression.REF_VAR) {
                // expr is a variable reference expression
                // gets the variable to which expr refers to
                Variable var = (Variable) expr.getReferree();

                if (var != null && var.getSort() == Variable.MEMBER) {
                    // var is a member variable
                    int lrsort = expr.getLRSort();
                    if (lrsort == Expression.R_VALUE) {
                        // var is use
                        if (!used.contains(var) && !defined.contains(var)) {
                            used.addElement(var);
                        }
                    }
                    else if (lrsort == Expression.L_VALUE
                             || lrsort == Expression.LR_VALUE) {
                        // var is definition
                        if (!defined.contains(var)) {
                            defined.addElement(var);
                        }
                    }
                }
            }
        }
    }

    /** Do nothing */
    public void doAfterTraverse(FrjObject obj) {
    }
}

/**
 *  Print used and defined member variables
 */
public class PrintMembersAndMethods {
    public static void main(String args[]) {
        if (args.length == 0) {
            System.err.println("usage: command ClassName");
            return;
        }

        // initialize
        ....

        Enumeration routines
            = Extents.getObjects(Routine.CLASS_NAME);
        // Routine.CLASS_NAME is a constant holding "Routine".

        while (routines.hasMoreElements()) {
            Routine routine = (Routine) routines.nextElement();

            // Traverse routine
            GetMembers tm = new GetMembers();
            routine.traverse(tm);

            // Print used and defined member variables
            System.out.println("Used variables in " + routine);
            printVars(tm.used);
            System.out.println("Defined variables in " + routine);
            printVars(tm.defined);
        }
    }
    ...
}
```

Figure 2.13: Example of TraverseMethod Class

In many relational database systems, a user can define an external schema (view). However, many object-oriented database systems do not support a view definition. We propose the view definition mechanism for the software database.

Defining a view on the object-oriented database is performed at two levels: the schema level (schema updates) and the object level (object updates)[4]. At the schema level, we define a new class hierarchy[11]. When users want to specialize a class, they can define new classes inherited from an original class. In the following program, the `Statement` class is specialized to the `Repetition` and `Selection` classes. The `Repetition` class is specialized to the `While` and `For` classes, too.

```
/*--- Original J-model class ------------*/
class Statement {
   ...
}

/*--- User defined view class ------------*/
/** Represents while-, do-, and for-statements */
class Repetition extends Statement {
   Expression getCondition() {...}
   Statement getBody() {...}
   ...
}

/** Represents "while (condition) body" statement */
class While extends Repetition {
   ...
}

/**
 * Represents "for (init; condition; update) body"
 *  statement
 */
class For extends Repetition {
   Expression getInitExpression() {...}
   Expression getUpdateExpression() {...}
   ...
}

/** Represents if-statement */
class Selection extends Statement {
   Expression getCondition() {...}
   Statement getThenPart() {...}
   Statement getElsePart() {...}   ...
}
```

[11] Classes defining a new class hierarchy are called *view classes*.

When users want to generalize a class, they can combine some original classes into a new class that has objects of original classes. A method of the new class is delegated to an appropriate object. In the following program, the `Identifier` class has an alternative object of the `Variable` or the `Routine` class, and a method `getName()` of the `Identifier` class is delegated to a *non-null* object: the `getName()` of a non-null object is invoked.

```
/** combines the Variable and the Routine class */
class Identifier {
   // an alternative object
   Variable variable;
   Routine routine;

   String getName() {
      // delegates getName() to an appropriate object
      if (variable != null) {
         return variable.getName();
      } else if (routine != null) {
         return routine.getName();
      } else {
         return null;
      }
   }
   ...
}
```

Note that this approach allows to define new view classes not only from classes of the access library but also from existing another view classes.

There are two consistencies about schema updates[4].

1. *Structural consistency* refers to the static part of an object-oriented database. A schema is structurally consistent if a class hierarchy is a tree, and if attributes of a class and a method (for example, its name, scope, and signature) are all compatible.

2. *Behavioral consistency* refers to the dynamic part of an object-oriented database. An object-oriented database is behaviorally consistent if each method respects its signature and if its code does not result in run-time errors or does not produce unexpected results.

Our approach is structurally consistent, because view classes that are structurally inconsistent cause a compiler's errors. However, Our approach does not ensure behaviorally consistent. Users must define view classes carefully to maintain behavioral consistency.

At the object level, we change specific objects dynamically. Because the access library retrieves data from the SDB according to J-model classes, generated objects are not instances of view classes. In Japid, this problem is solved in the two ways.

1. At instantiation, data of the SDB are translated into objects according to view classes instead of J-model classes, or

2. After instantiation of J-model classes, we create objects of view classes, copy attributes from J-model objects to them, and replace J-model objects and its associations with new one (Figure 2.14).

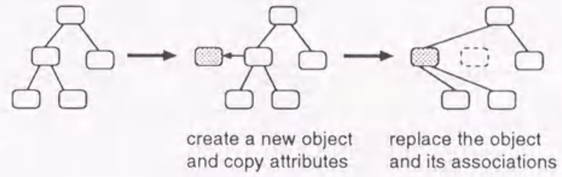


Figure 2.14: Object Updates

2.2.3 Changing Program

Japid supports changing source programs. We can consider changing programs as manipulating objects in the SDB. It enables manipulations such as changing a variable's name consistently. For example, in a sample program in Figure 2.6, when we want to change a parameter variable a's name, we must change the right-hand side in line 4, too. But we must not change variables a in line 2 and the left-hand side in line 4. An idea for consistent changes of source programs has already been proposed[15]. In this section, we explain this idea and its implementation with Japid.

The source program analysis by a compiler is classified into the three stages: the lexical, the syntactic, and the semantic stages. At the lexical stage, a compiler recognizes tokens. At the syntactic stage, it reduces tokens to symbols and checks whether source programs satisfy the grammar. At the semantic stage, it checks semantics of programs: for example, whether an operator is applied to operands appropriately, whether no variable is used without declaration, and so on. Source programs are translated into abstract syntax trees or semantic graphs through these three stages.

In Japid, users can change source programs by manipulating objects of semantic graphs managed by the SDB. It is natural that objects are translated into programs through the three stages: the semantic, the syntactic, and the lexical stages. This is the reverse order of a compiler's three stages. First, the manipulation is checked semantically. For example, in changing a variable's name, a name must not conflict with other variables' names. Second, the manipulation is checked syntactically. In the case of changing a variable's name, this stage is passed because the structure of an abstract syntax tree is not changed. Finally, at the lexical stage, the manipulation is checked lexically and source programs are changed actually by the writer. For example, a variable's name must start with an alphabet or an underscore ('_') and must not include an illegal character which is not an alphabet, a digit, nor an underscore.

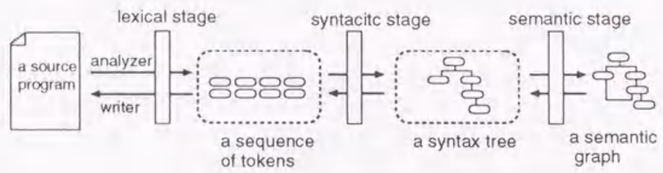


Figure 2.15: Three Stages

We can realize this idea by a class hierarchy with three layers using view definition mechanism (Figure 2.16). The top layer consists of lexical classes inherited from J-model classes. The middle layer consists of syntactic classes inherited from lexical classes. The bottom layer consists of semantic classes inherited from syntactic classes. Users manipulate semantic objects, and its manipulations are propagated to upper classes. When users want to define new constraints, they can define classes inherited from semantic classes and define methods that satisfy their constraints.

We show a sample program that changes a variable's name in Figure 2.17. In Figure 2.17, LexicalVariable, SyntaxVariable, and SemanticVariable classes are defined as view classes. When a user invokes the `SemanticVariable#changeName(String)` method, it checks semantic constraints and invokes `SyntaxVariable#changeName(String)`. It checks syntactic constraints and invokes `LexicalVariable#changeName(String)`. It checks lexical constraints and invokes `JapidObject#changeToken(String)` to modify the name attribute of this object actually. After changing a variable name, `SemanticVariable#changeName(String)` also changes expressions referring to this variable.

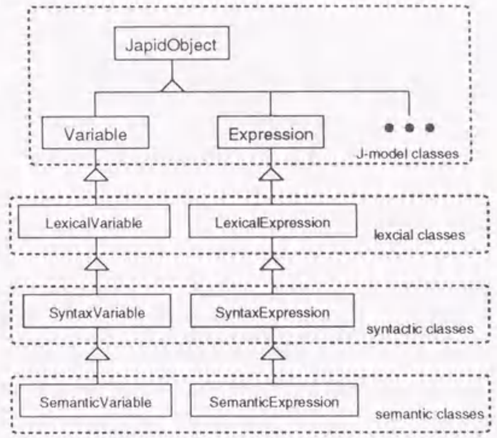


Figure 2.16: Three Layers

Let us take another example. We show constraints for inserting statements into a program. This change is often used for debugging and for analyzing execution of a program: *print-statements* such as "`System.err.println("val1: " + val1);`" are inserted for tracing a program execution and for logging a program's states.

Semantic constraints of this change are follows:

1. Variables and methods used in inserted statements must be declared. Especially, local variables must be declared in precedent statements.

2. All expressions included in statements satisfy semantic constraints for the expression. For example, an operator must be applied to appropriate operands: when `n` is an integer variable and `flag` is a boolean variable, the statement "`n = n + 2;`" satisfies this constraint, but the statement "`n = flag;`" does not.

A syntactic constraint is that inserted statements must be parsed without syntax errors. In Japid, semantic graphs must be appropriate: objects must be associated with appropriate objects.

Lexical constraints are follows:

1. A token separator (e.g., a white space) must be put between successive identifiers. For example, in the variable declaration, a separator must be put between a type and a variable: the statement "`String str;`" satisfies this constraint, but the statement "`Stringstr;`" does not.

```
1: /** The super class of all classes of the access library */
2: class JapidObject {
3:     /** Change the token represented by this object.
4:       * This modification is reflected on source programs by the writer.
5:       */
6:     void changeToken(String name) {
7:         ....
8:     }
9: }
10:
11: class Variable extends JapidObject {
12:     ...
13: }
14:
15: class LexicalVariable extends Variable {
16:     void changeName(String name) throws LexicalErrorException {
17:         if (!isValidIdentifier(name)) {
18:             throw new LexicalErrorException("Illegal Character");
19:         }
20:         // The lexical check is passed. Change program.
21:         super.changeToken(name);
22:     }
23:     // Checks whether a name satisfy an identifier rule.
24:     boolean isValidIdentifier(String name) {
25:         ...
26:     }
27: }
28:
29: class SyntaxVariable extends LexicalVariable {
30:     void changeName(String name) {
31:         // The syntactic check about changing name is nothing
32:         super.changeName(name);
33:     }
34: }
35:
36: class SemanticVariable extends SyntaxVariable {
37:    void changeName(String name)
38:        throws LexicalErrorException, SemanticErrorException {
39:        // Is there a variable which has the same name ?
40:        for (Enumeration e = getVariablesInThisScope();
41:             e.hasMoreElements();) {
42:            Variable var = (Variable) e.nextElement();
43:            if (var.getName().equals(name)) {
44:                throw new SemanticErrorException("Name Conflict");
45:            }
46:        }
47:        // The semantic check is passed.
48:        // Checks syntacically and change progrmas.
49:        try {
50:            super.changeName(name);
51:            // Change expressions referring this variable.
52:            for (Enumeration e = getReferers();
53:                 e.hasMoreElements();) {
54:                SemanticExpression expr
55:                    = (SemanticExpression) e.nextElement();
56:                expr.changeVariableName(name);
57:            }
58:        }
59:        catch (Exception e) {
60:            // Error recovery
61:        }
62:    }
63: }
```

Figure 2.17: Classes for Changing Variable's Name

2. A coding style (e.g., K&R style and GNU style) of inserted statements matches one of other statements. This constraint is important for readability, but this is not essential for compilation. In the case that we do not have to understand changed programs, for example, in specialization, we can ignore this constraint.

2.3 Evaluation

We implemented our system. The analyzer is written in C, lex, and yacc (about 6500 lines). The access library is written in Java (54 classes, about 9000 lines). A writer can be implemented in two ways: modifying the original source programs and generating new source programs that have the same functions as originals. We implemented a writer in the latter way using the view definition mechanism (in Java, 15 classes, about 1200 lines). It traverses objects along syntax associations and translates them into source programs. We confirmed that new generated programs could be compiled without errors, and compiled programs behaved in the same way as original programs for some sample inputs. We have also been implementing a writer in the former way. Implementing the program change mechanism has not been completed. We implemented some view classes for changing the name of an identifier.

2.3.1 Performance

We measured the execution time at Sun Ultra1 OEM workstation (300 MHz UltraSPARC-II, 128MB RAM). Programs written in Java were interpreted by JDK-1.1.6's Java virtual machine[12].

First, we measured performance of each part of Japid. A target source program was the `java.lang.String` class (1531 lines). It was translated into 3882 objects and 4546 associations. Table 2.1 shows Execution time of each part of Japid.

Second, we measured performance about dynamic linking. In Java, each expression has its type. Table 2.2 shows five expressions and their types of a statement "`d = Math.sqrt(a + 1.0);`", when let `a` be an integer variable and `d` be a double variable. In Japid, this association is represented by the *has_type* association between the *Expression* and the *Type* classes. The analyzer, however, does not produce it because it is an association between classes. In this example, the method invocation expression `Math.sqrt` needs

[12]Programs interpreted by a JIT (Just In Time) compiler run about twice faster than by JDK's virtual machine.

Table 2.1: Execution Time (String)

Function (part)	Time[sec.]
analyzing program (analyzer)	0.6
translating data of SDB into objects (access library)	7.6
translating objects into a source program (writer)	1.3

NOTE: The time of the writer does not include the time of loading the SDB.

information of the another class `Math`. *has_type* associations are produced by the tool, called `SetExprType`, which loads and links fine grained databases and sets expressions' types [13]. `SetExprType` loads all data from the fine grained database of the target class and loads only necessary data of classes used by the target class. Necessary data are objects of some J-model classes such as *Routine*, *Variable*, *Type*, and so on[14].

We measured the execution time of `SetExprType`. A target program was the `ImageMap` class which is a demo program of JDK-1.1.6. It had 453 lines and was translated into 1643 objects (objects of Expression class were 666) and 1870 associations. It used 34 classes. `SetExprType` loaded databases[15], linked them and saved data to databases in 55.0 seconds.

Table 2.2: Expressions and Their Types

Expression	Type
`d = Math.sqrt(a + 1.0)`	double
`d`	double
`Math.sqrt(a + 1.0)`	double
`a + 1.0`	double
`a`	int
`1.0`	double

Generally, at a fine grained level, a program does not get good performance, because of overhead of creating and managing many objects. But, we could get performance enough to use practically.

[13] An expression's type is set using `TraverseMethod#doAfterTraverse(JapidObject)`. For example, a type of an expression `a+1.0` can be set after types of expressions `a` and 1.0 are set. `doAfterTraverse(JapidObject)` is suitable for doing them.

[14] It is a kind of user defined view.

[15] Finally it loaded 35 fine grained databases, 13491 objects, and 9969 associations.

2.3.2 CASE Tools with Japid

Some CASE tools have been developed using Japid. In this section, we show them. They could be written in a few codes. We confirm our system provides sufficient API for writing them.

Object Diagram Editor

An object diagram of OMT[30] is very useful for modeling a real world and for understanding structure of existing programs. We wrote an editor for an object diagram (about 2800 lines, Figure 2.18). We can draw an object diagram using it. It translates a diagram to templates of source programs[16]. It also translates Java source programs to an object diagram. Figure 2.18 shows an object diagram of some classes in *java.io* package.

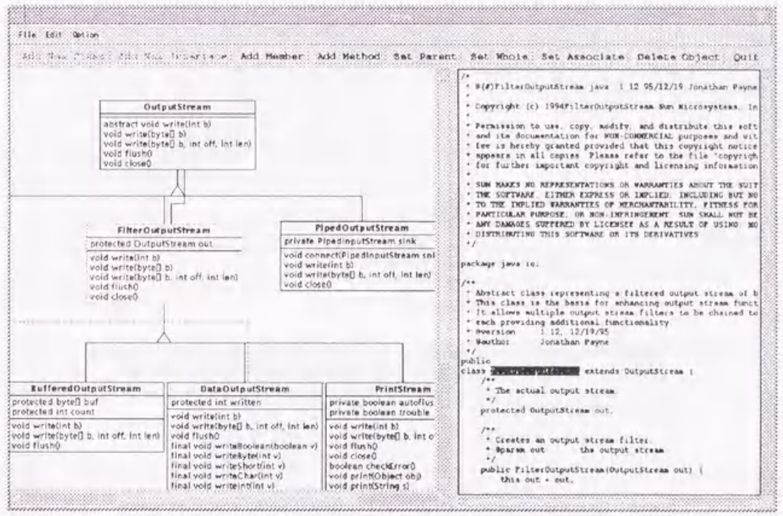


Figure 2.18: Object Diagram Editor

Identifier Name Changer

On programming, we often want to change the name of a variable or a method for some reasons such as a change of specifications, improving understand-

[16] A template of a program consists of classes, member variables, and methods without bodies.

ability, correcting spelling, and so on. But changing a name is very difficult because it requires analyzing source programs syntactically and semantically.

A tool that performs the consistent change of the name of an identifier was developed using our program change mechanism (about 5000 lines, including lexical, syntax, and semantic classes, Figure 2.19). It can change the name of a class, a method, and a variable. In the case of changing the name of a class, the names of constructors are also changed. In the case of changing the name of a method, when the method has overridden or overloaded methods, a tool user can choose changing their names or not.

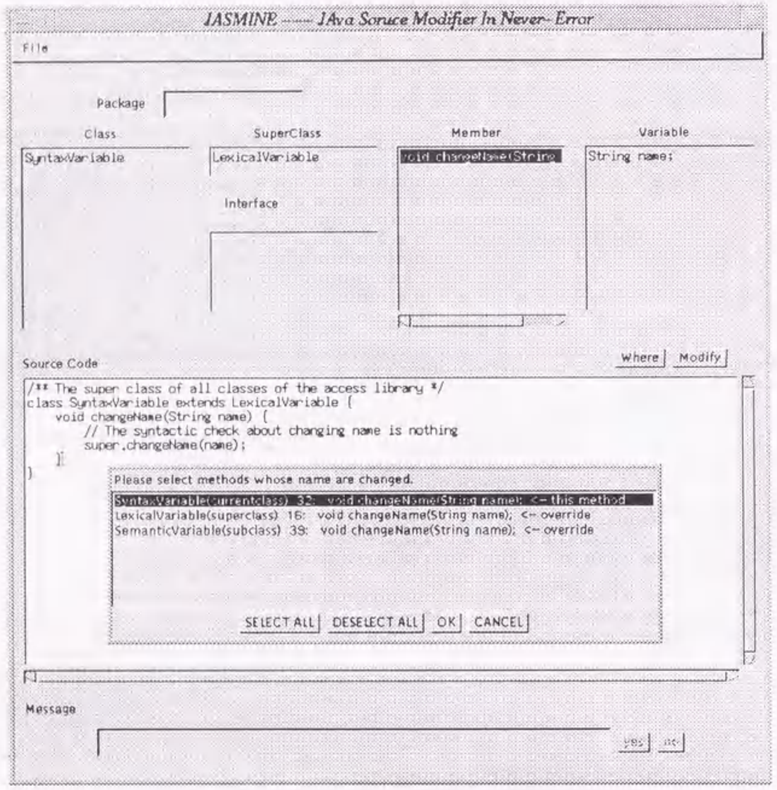


Figure 2.19: Identifier Name Changer

Method Name Completion

A long name of a method such as `URLConnection#guessContentTypeFromStream()` is useful for program understanding. But, on writing a program, we often forget a precise method's name and misspell it.

A tool that complements the name of a method was developed on Emacs editor (about 990 lines in Emacs Lisp and about 260 lines in Java).

Figure 2.20 shows possible completions at `din.read`. This tool recognizes that `din` is an instance of the class `BufferedReader` and method's name starts with `read`. It shows a list of the name and parameters of methods. It also recognizes that the `BufferedReader` is a subclass of the `Reader` and shows methods of the class `Reader`.

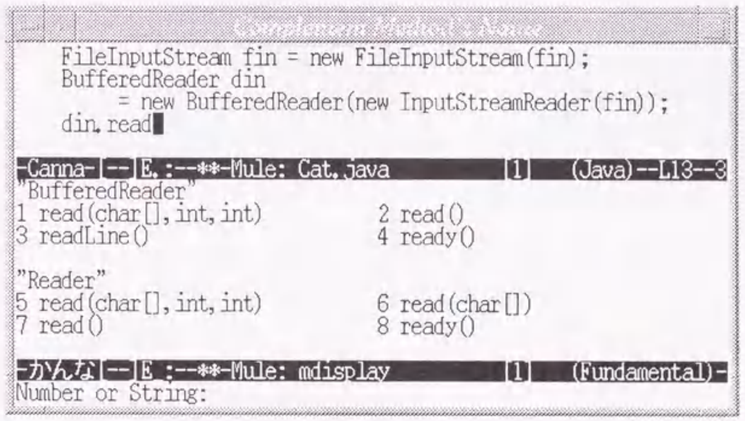


Figure 2.20: Method's Name Completion

Browser for Objects and Links in SDB

Developers who define their own views will want to see their objects and links for debugging and testing. We developed a browser which shows data structure of the software database according to J-model classes or to user defined view classes (about 670 lines, Figure 2.21).

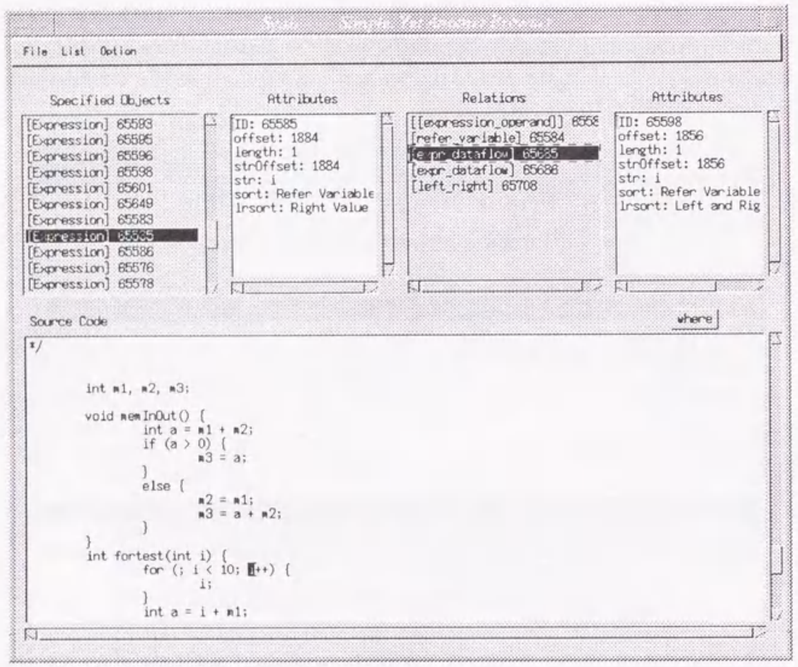


Figure 2.21: Browser for Objects and Links in SDB

2.4 Summary

In this chapter, we have proposed Japid, a CASE tool platform for Java. We have shown the view definition and program change mechanisms. We implemented our system, and confirmed its effectiveness; we could get performance enough to use practically, we could write CASE tools in a few codes, and our system provides sufficient API for developing CASE tools. Our system enables changing the name of an identifier consistently. It is very difficult for tools at the lexical and the syntactic levels, because source programs must be changed at the semantic level.

We have shown constraints for changing the name of an identifier and for inserting statements, and created a tool that performs the consistent change of the name of an identifier. In future, we plan to clarify constraints of other changes and to implement the program change mechanism completely.

Chapter 3

Object-Oriented System Dependence Graph

In this chapter, to achieve the research goals (2) and (3), we propose new dependence analysis for object-oriented programs. Dependence analysis is important for program optimization, understanding and so on. Many dependence analysis methods for procedural programs have been proposed[20][13][21]. However, few techniques for object-oriented programs have been proposed [23][24].

Krishnaswamy proposed the object-oriented program dependency graph (OPDG) based on the program dependence graph for procedural programs[23][28]. The OPDG consists of three parts: the class hierarchy subgraph, the control dependence subgraph, and the data dependence subgraph. However, the data dependence subgraph is not defined completely.

Larsen and Harrold accommodated the system dependence graph, which is proposed in [20], to object-oriented features such as inheritance and dynamic method call [24]. It is very useful, but some problems are left; it does not represent method composition, it violates capsulation of information, and so on.

In this chapter, we propose a new dependence graph, the **Object-Oriented System Dependence Graph (OSDG)**. In Section 3.1, we explain some definitions for dependence analysis. In Section 3.2, we explain the system dependence graph for object-oriented software (SDG) proposed by Larsen and Harrold, and show its problems. To solve them, we propose a new dependence graph, the OSDG in Section 3.3. The OSDG is built at the *expression-grain* level for providing more precise information than graphs proposed previously. It can represent data dependences of method composition such as `f(g(a+1, b), obj.h(0))` and of a statement that has multiple

occurrences of the same variable such as `a = f(a) + g(a);`. Generally, analysis at a fine grain level produces the large amount of data. We also propose a method for decreasing the amount of nodes and edges by changing granularity of a graph: we translate a graph that consists of *expression* nodes into a graph that consists of *statement* nodes or of *method* nodes. In Section 3.4, we show an implementation of the OSDG using Japid and its performance. In Section 3.5, we show application of the OSDG, object dependence analysis and program slicing.

3.1 Preparation

In this section, we explain some definitions for dependence analysis [26][38].

Definition 3.1 (Control Flow) A control flow shows the sequence of execution of statements in a program. When the execution of a statement S_1 precedes the execution of a statement S_2, we say there is a **control flow** from S_1 to S_2. □

Definition 3.2 (Control Dependence) When the execution of a statement S_2 is dominated by the result of the execution of a statement S_1, we say there is a **control dependence** from S_1 to S_2. □

Definition 3.3 (Data Flow) When all of the following conditions are satisfied, we say there is a **data flow** for a variable x from a statement S_1 to a statement S_2.

1. The statement S_1 *defines* the variable x: a value is assigned to x in S_1.
2. The statement S_2 *uses* the variable x: only the value of x is referred in S_2, i.e., a value is not assigned to x.
3. There is a control flow path from S_1 to S_2 and the variable x is not defined in statements included in its path. □

Definition 3.4 (Data Dependence) When there is a data flow from a statement S_1 to a statement S_2, we say there is a **true data dependence** from S_1 to S_2. In this thesis, it is also called a **data dependence**, simply. □

We show an example of these relationships in Figure 3.1.

```
S1: sum = 1;
S2: i = 1;
S3: while (i < 11) {
S4:     sum = sum + 1;
S5:     i = i + 1;
    }
```

control flow	(S1, S2), (S2, S3), (S3, S4), (S4, S5), (S5, S3)
control dependence	(S3, S4), (S3, S5)
data flow	(S1, S4), (S4, S4), (S2, S3), (S2, S5), (S5, S3), (S5, S5)
data dependence	(S1, S4), (S4, S4), (S2, S3), (S2, S5), (S5, S3), (S5, S5)

Figure 3.1: Example of Flow and Dependence

3.2 System Dependence Graph for Object-Oriented Software

In this section, we explain the system dependence graph for object-oriented software (SDG), which is proposed in [24], and we show its problems.

3.2.1 Class Dependence Graph

A SDG consists of class dependence graphs (ClDG). A ClDG is a directed graph whose root node represents a class. Its nodes are classified into six kinds.

- A *class entry* node (C) represents a class.
- A *method entry* node (M) represents a method.
- A *statement* node (S) represents a statement.
- A *call* node ($CALL$) represents a method call in a statement.
- A *parameter* node (P) represents a parameter of a method. It is classified into four kinds.
  - *formal_in* represents a formal parameter of a method or a member variable used in the method.
  - *formal_out* represents a call-by-reference formal parameter of a method or a member variable defined in the method.
  - *actual_in* represents an actual parameter of a method call.

  - *actual_out* represents a call-by-reference actual parameter of a method call.
- A *polymorphic choice* node (PC) represents a dynamic method call.

Edges of ClDG are classified into six kinds.

- A *class member* edge ($C \times M$) represents a *has-a* association: a class has a method.
- A *control dependence* edge ($M \times S, S \times S, S \times CALL, S \times PC, CALL \times P$) represents a control dependence. In the SDG, an association between a method call and its actual parameter is represented as a control dependence.
- A *data dependence* edge ($S \times S, P \times S, S \times P$) represents a data dependence.
- A *call* edge ($CALL \times M, PC \times CALL$) represents method call.
- A *parameter* edge ($P \times P$) represents an association between formal and actual parameters.
- A *summary* edge ($P \times P$) represents a data dependence from an *actual_in* node to an *actual_out* node.

The SDG of a program is built by composing ClDGs. We show an example of a SDG in Figure 3.2.

3.2.2 Graph of Derived Class

The ClDG of a derived class is built by composing class entry nodes of it and method entry nodes of its base class; method entry edges from a class entry node of a derived class to method entry nodes of a base class are created.

We show an example of the SDG of a derived class in Figure 3.3. Subgraphs of methods' bodies are omitted in Figure 3.3.

3.2.3 Graph for Method Call

When a statement calls a method, a subgraph is created in the following way.

1. A call node is created.
2. *actual_in* and *actual_out* nodes are created.

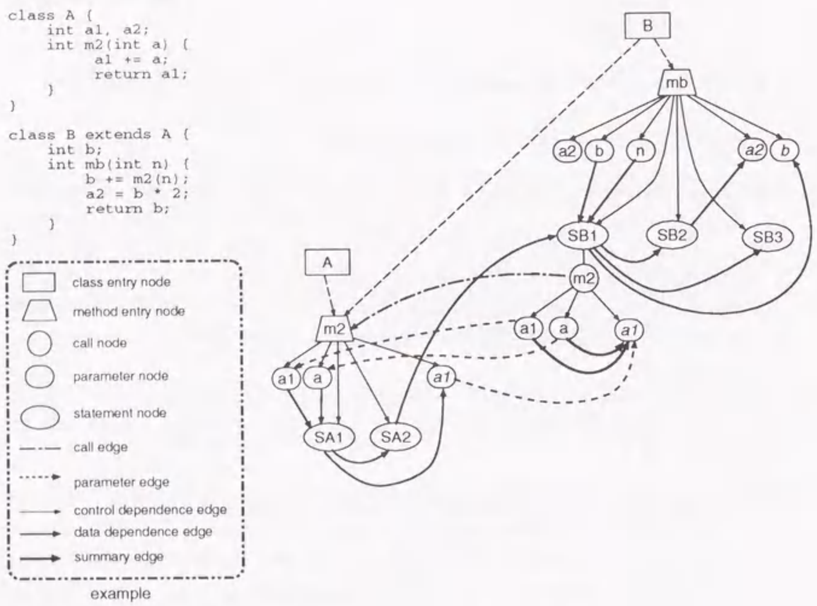


Figure 3.2: Example of SDG

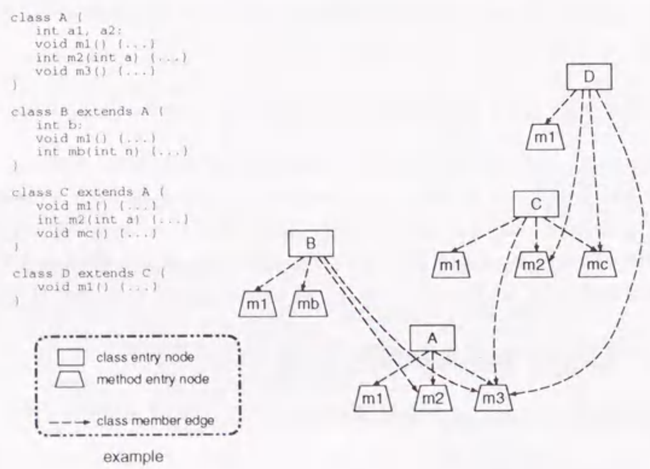


Figure 3.3: SDG between Classes and Methods

3. Following control dependence edges are created:
   - from the statement node to the call node, which is created in 1.
   - from the call node to *actual_in* and *actual_out* nodes, which are created in 2.
4. A call edge from the call node to the method entry node is created.
5. Parameter edges from *actual_in* to *formal_in* and from *formal_out* to *actual_out* are created. When the method returns a value, a data dependence edge from a statement node that shows a return-statement to the statement node that has a method call, is created.

When a statement has a dynamic method call, a polymorphic choice node is created, and following call edges are created:

- from the statement node that has method call to the polymorphic choice node
- from the polymorphic choice node to call nodes that may be called

We show an example of a SDG having dynamic method call in Figure 3.4.

3.2.4 Evaluation of SDG

The SDG has some advantages; it defines a way to pass parameters on method call and to represent a dynamic method call. It, however, has following disadvantages.

1. It cannot represent method composition such as `f(g(a+1, b), obj.h(0)))`.
2. It violates capsulation of information, which is a key feature of object-oriented programs; when new methods are added to a base class, the ClDG of a derived class must be changed. When data dependences between formal parameters of a method are changed, summary edges must be changed, too. It causes low reusability.
3. It cannot represent referring to member variables from external classes.

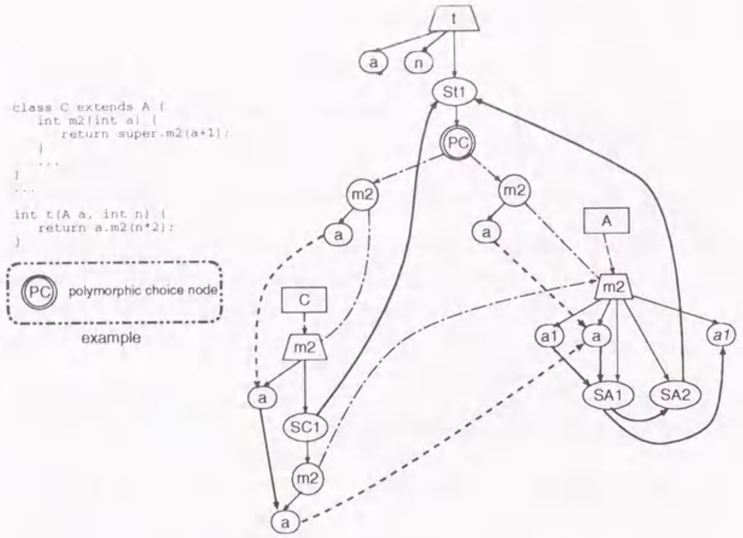


Figure 3.4: SDG for Dynamic Method Call

3.3 Object-Oriented System Dependence Graph

In this section, to solve problems of the SDG, we propose a new dependence graph, **Object-Oriented System Dependence Graph**, **OSDG**. It is built using an abstract syntax tree at the *expression-grain* level, in order to provide more precise information.

We show an overview of the OSDG from Section 3.3.1 to 3.3.4, and we give definition of the OSDG in Section 3.3.5. The OSDG increases the amount of data because it is built at the expression-grain level. We propose a method for decreasing the amount of data by changing granularity of a graph in Section 3.3.6.

3.3.1 Subgraph between Classes, Methods, and Member Variables

To solve some problems of the SDG, we add a *member variable* node and *class inheritance* edge to the SDG. A *member variable* node represents a member variable. A *class member* edge from a *class entry* node to a *member variable* node is created. It is used for direct access to a member variable

from external classes.

A *class inheritance* edge from the *class entry* node of a derived class to one of the base class is created. In the SDG, when a new method is added to the base class, a class member edge from the class entry node of a derived class to the method entry node of the new method must be created. In the OSDG, however, we do not need to create a *class member* edge in this case.

We show a subgraph between classes, methods, and member variables in Figure 3.5

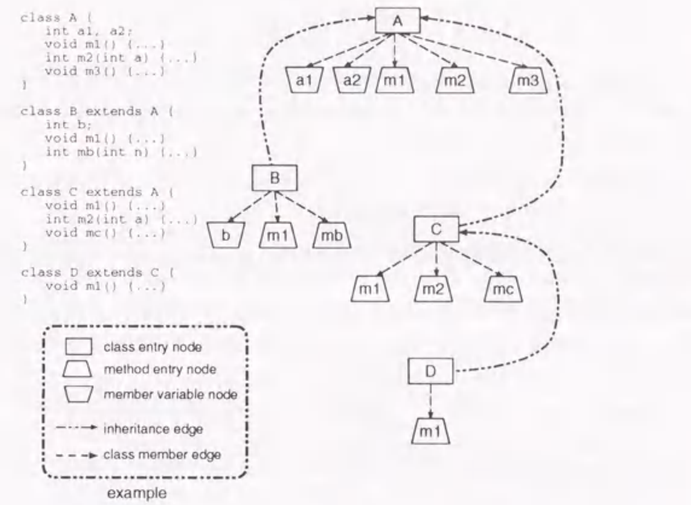


Figure 3.5: OSDG between Classes, Methods, and Member Variables

3.3.2 Control Dependence

In the OSDG, a control dependence is an association between statements, and between a method and statements that compose the body of the method.

3.3.3 Data Dependence

Many dependence graphs such as the SDG and the PDG cannot show data dependences of method composition such as `f(g(a+1, b), obj.h(0))`.

Furthermore, in traditional dependence graphs of a program in Figure 3.6, two data dependence edges for variable `a`, (S1, S2) and (S2, S3) are made. But, because variable `a` occurs three times in the statement S2, we do

not understand which a in S2 is dependent on S1 from its graph. When the methods f and g are call-by-value, both a of f(a) and g(a) are dependent on S1. When the method f is call-by-reference, and + is a left-associative operator, a of f(a) is dependent on S1 and a of g(a) is dependent on f(a). Traditional graphs cannot represent and distinguish these dependences.

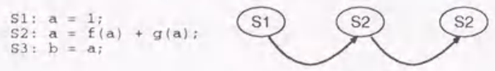


Figure 3.6: Sample Program

These problems are caused by data dependence analysis at the *statement-grain* level. In the OSDG, to solve them, a data dependence is represented at the *expression-grain* level.

Data Dependence between Expressions

We define some definitions for a data dependence between expressions.

Definition 3.5 (Elementary Expression) We call an expression that does not consist of some other expressions an **elementary expression**. For example, a=b+1 has five expressions, a=b+1, a, b+1, b, and 1, and three elementary expressions, a, b, and 1. □

Definition 3.6 (Variable Reference Expression) When an expression E defines or uses a variable and E is an elementary expression, we call E a **variable reference expression**. For example, a=b+1 has two variable reference expressions, a and b. □

We define a data flow and a data dependence between expressions as follows.

Definition 3.7 (Data Flow between Expressions) When variable reference expressions E_1 and E_2 satisfy all of the following conditions, we say there is a **data flow** for a variable x from E_1 to E_2.

1. The expression E_1 defines the variable x.
2. In the statement S_1, which includes the expression E_1, the variable x is not defined after evaluating the expression E_1.
3. The expression E_2 uses the variable x.

4. In the statement S_2, which includes the expression E_2, the variable x is not defined before evaluating the expression E_2.
5. There is a control flow path from S_1 to S_2 and the variable x is not defined in statements included in its path. □

Definition 3.8 (Data Dependence between Expressions) When variable reference expressions E_1 and E_2 satisfy one of the following conditions, we say there is a **data dependence** from E_1 to E_2.

1. There is a data flow from E_1 to E_2.
2. There is an assignment expression such that E_1 is a right-hand side and E_2 is a left-hand side. □

In this thesis, "$E_1 \xrightarrow{d} E_2$" represents a data dependence from expression E_1 to E_2.

We show examples of data dependences between expressions in Figure 3.7. In this figure, methods f and g are call-by-value. A node labeled "ObjE" represents an operation on an object such as obj.mem and a node labeled "Expr" represents a composite expression such as a+b.

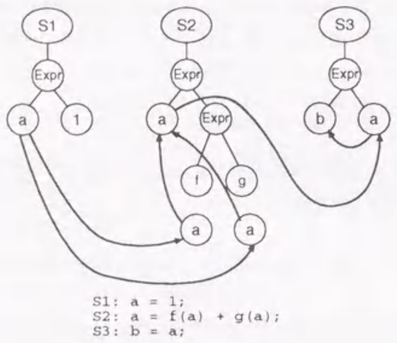
3.3.4 Graph for Method Call

The OSDG represents a method call as an expression node. Because a method call is analyzed at the expression-grain level, method composition such as f(g(a+1, b), obj.h(0))) and an operation on an object such as obj.m() are represented correctly.

Parameter nodes

We classify a parameter node into eight kinds.

1. *formal_in* represents a formal parameter of a method.
2. *member_in* represents a member variable used in a method.
3. *formal_out* represents a call-by-reference formal parameter.
4. *member_out* represents a member variable defined in a method.
5. *return* represents a value that a method returns.
6. *actual_in* represents an actual parameter on a method call.



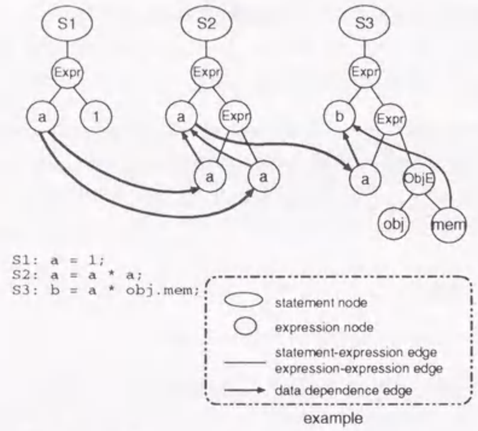


Figure 3.7: Data Dependence Graph at Expression-Grain Level

7. *actual_out* represents a call-by-reference actual parameter on a method call.

8. *returned* represents a value returned from a method.

In the OSDG, *formal_in* and *formal_out* nodes represent only formal parameters. Member variables are represented by *member_in* and *member_out*. They are used for direct access to a member variable not only within the same class but also from external classes.

We also add new nodes, *return* and *returned*. A method returns a value by the *return* node and a method call expression received a value from the *returned* node. In the OSDG, a return-statement "`return EXPR;`" is represented as "`return (`*ret* `= EXPR);`": *ret* is a *virtual* variable.

Constructing Graph for Method Call

An abstract syntax subtree that shows a method call is translated into the OSDG as follows:

1. *actual_in*, *actual_out* and *returned* nodes are created.

2. Edges from a method call expression node to *actual_in*, *actual_out*, and *returned* nodes are created. They are called *method parameter* edges.

3. Edges from expression nodes passed to a method as actual parameters to *actual_in* and *actual_out* nodes are created. They are called *parameter expression* edges.

4. An edge from a method call expression node to a method entry node is created. It is called a *method call* edge.

5. Edges from *actual_in* to *formal_in*, from *formal_out* to *actual_out*, and from *return* to *returned* are created. They are called *parameter passing* edges.

Parameter passing edges also represents data dependences such as *actual_in* $\xrightarrow{d}$ *formal_in*, *formal_out* $\xrightarrow{d}$ *actual_out*, and *return* $\xrightarrow{d}$ *returned*.

We show examples of static method calls in Figures 3.8 and 3.9. For example, from Figure 3.8, we understand that a return value of a method `mb` is dependent on `n` (a parameter of a method mb), `a1` (a member variable of class `A`), and `b` (a member variable of class `B`).

In Figure 3.9, returned values from the methods `g` and `h` are passed to the method `f` as actual parameters. Edges from *formal_in* to *formal_out* such as

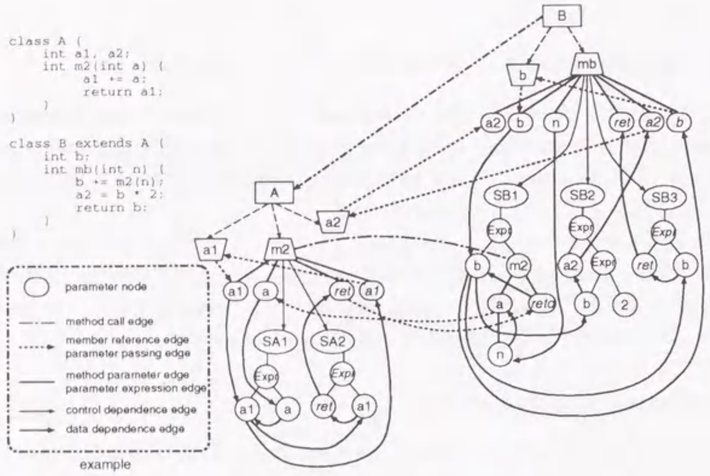


Figure 3.8: Example of OSDG

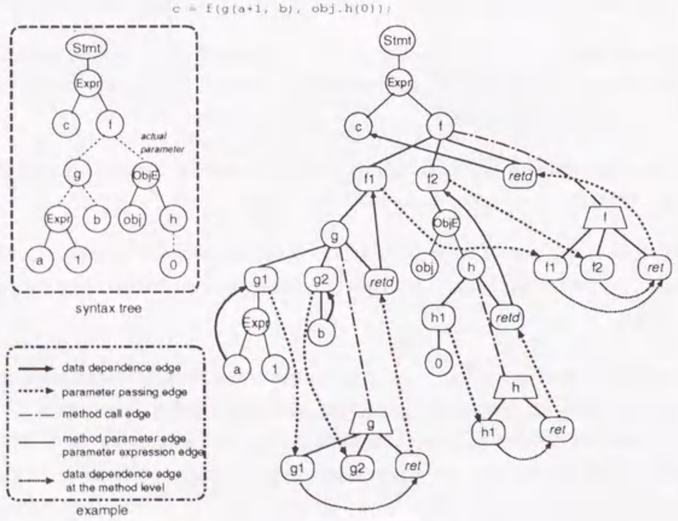


Figure 3.9: OSDG for Method Composition

(g1, ret) and (h1, ret) represent data dependences between formal parameters (see Section 3.3.6).

A dynamic method call is represented by a *polymorphic choice* node like the SDG (Figure 3.10).

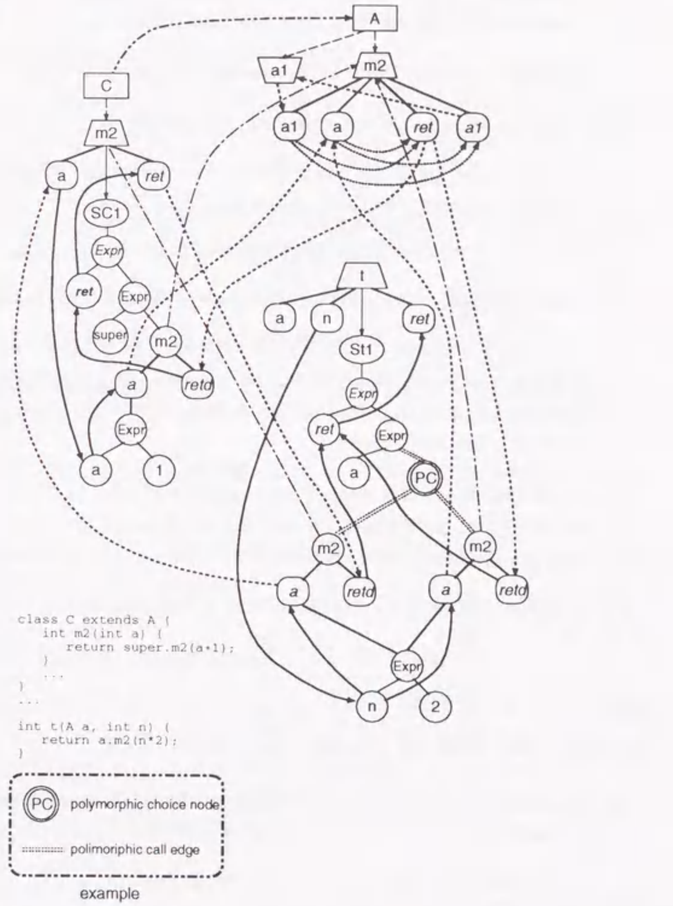


Figure 3.10: OSDG for Dynamic Method Call

3.3.5 Definition of OSDG

In this section, we define the OSDG. It is constructed using an abstract syntax tree.

Node

A Node of the OSDG is classified into eight kinds.

1. A *class entry* node (C) represents a class.
2. A *method entry* node (M) represents a method.
3. A *member variable* node (V) represents a member variable.
4. A *statement* node (S) represents a statement.
5. An *expression* node (E) represents an expression. Especially, an expression showing a method call is labeled $CALL$.

   These five nodes occur in an abstract syntax tree.
6. A *polymorphic choice* node (PC) represents a dynamic method call.
7. A *formal parameter* node (P_f) is classified into three kinds: a formal parameter of a method (*formal_in, formal_out*), a member variable to which is referred in a method (*member_in, member_out*), and a return value of a method (*return*).
8. An *actual parameter* node (P_a) is classified into two kinds: an actual parameter of a method (*actual_in, actual_out*) and a value returned from a called method (*returned*).

We call a set of formal parameter and actual parameter nodes *parameter* nodes (P).

Edge

An edge of the OSDG is classified into thirteen kinds.

1. A *class inheritance* edge ($C \times C$) represents an inheritance association between classes.
2. A *class member* edge ($C \times M, C \times V$) represents a *has-a* association: a class has a method and a variable.
3. A *statement-expression* edge ($S \times E$) represents a *consist-of* association: a statement consists of expressions.
4. An *expression-expression* edge ($E \times E$) represents a *consist-of* association: an expression consists of expressions.

   These four edges occur in an abstract syntax tree.

5. A *member reference* edge ($P_f \times V$) represents a reference association: a formal parameter node (*member_in, member_out*) refers to a member variable.
6. A *method parameter* edge ($M \times P_f$, $CALL \times P_a$) represents an association between a method and one of its formal parameters or between a method call and one of its actual parameters.
7. A *parameter expression* edge ($P_a \times E$) represents an association between an actual parameter node and an expression node passed to a method as an actual parameter.
8. A *parameter passing* edge ($P \times P$) represents an association between an actual parameter and a formal parameter.
9. A *method call* edge ($CALL \times M$) represents a reference association between a method call expression and a called method.
10. A *polymorphic call* edge ($E \times PC, PC \times CALL$) represents a dynamic method call.
11. A *control dependence* edge ($M \times S, S \times S$) represents a control dependence.
12. A *data dependence* edge ($E \times E, P \times E, E \times P$) represents a data dependence between expressions.

3.3.6 Changing Granularity of Graph

In our approach, a dependence graph at *expression-grain* level has advantages and a disadvantage. An important advantage is that a data dependence is represented more precisely than traditional graphs. A disadvantage is that it increases nodes and edges. In general, there is a case that we do not need expression nodes and a case that we need only dependences between formal parameters of a method.

In this section, we propose a method translating an expression-grained graph into coarser grained graphs, a graph at the *statement-grain* level and a graph at the *method-grain* level. Owing to it, we can decrease the amount of nodes and edges of a graph. We can express our concerning point with a precise graph at the expression-grain level and other points with brief graphs at the statement-grain or the method-grain level. For example, in Figure 3.10, the methods `C#m2` and `t` are expressed at the expression-grain level and the method `A#m2` is expressed at the method-grain level.

Graph at Statement-Grain Level

We translate an OSDG, which is a graph at the expression-grain level, into a graph at the statement-grain level as follows.

1. For a variable reference expression e,
   (a) when e or an expression consisting of e is an actual parameter of a method call, we change start and end nodes of data dependence edges from e or to e into the actual parameter node.
   (b) Otherwise, we change start and end nodes of data dependence edges from e or to e into the statement node including e.
2. We create an edge between a method call expression node and the statement node including it. This edge represents that a statement has a method call.
3. We delete following nodes and edges from the original OSDG.
   - expression nodes other than method call expression nodes
   - statement-expression, expression-expression, parameter expression edges, and data dependence edges whose start or end point is an expression node.

Graph at Method-Grain Level

We translate an OSDG into a graph at the method-grain level as follows.

1. Let *in* be a formal parameter node that shows *formal_in* or *member_in* and let *out* be a formal parameter node showing *formal_out*, *member_out*, or *return*. When there is an *out* that is reachable from an *in* along data dependence and parameter passing edges, we create an edge from the *in* to the *out*. This edge represents a data dependence between formal parameter nodes.
2. We delete nodes and edges other than followings:
   - class entry, method entry, member variable, and formal parameter nodes
   - class inheritance, class member, method parameter, member reference edges, and edges created in 1.

We show a graph at the statement-grain level in Figure 3.11, and one at the method-grain level in Figure 3.12; these graphs are translated from Figure 3.8. Figures 3.9 and 3.10 are graphs composed of subgraphs at the expression-grain and at the method-grain level.

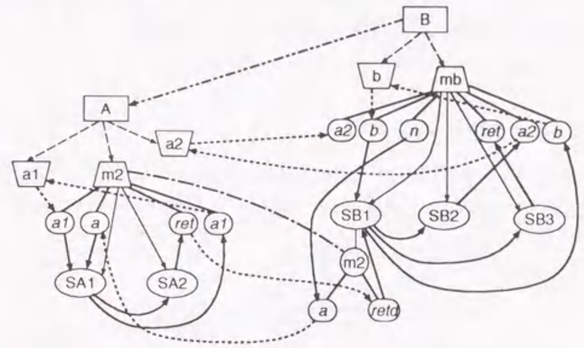


Figure 3.11: Graph at Statement Level

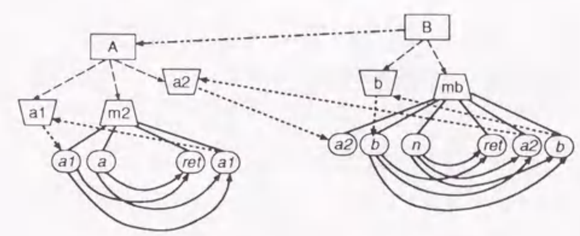


Figure 3.12: Graph at Method Level

3.4 Evaluation

3.4.1 Comparison with SDG

In the OSDG, problems of the SDG are solved as follows.

1. The OSDG can represent method composition and a statement having multiple occurrences of the same variable, because it is built at the expression-grain level.
2. The OSDG keeps capsulation of information. A class member edge between a derived class and a method of a base class and a summary

edge of the SDG are deleted. Instead of them, an inheritance edge is added to the OSDG, and a graph at the method-grain level can represent data dependences between formal parameters.

3. The OSDG has a member variable node and a member reference edge. They can represent a direct access to a member variable from external classes.

3.4.2 Implementation

In order to construct the OSDG actually, we have modeled it as a user defined view of Japid. We show a class diagram of the OSDG in Figure 3.13.

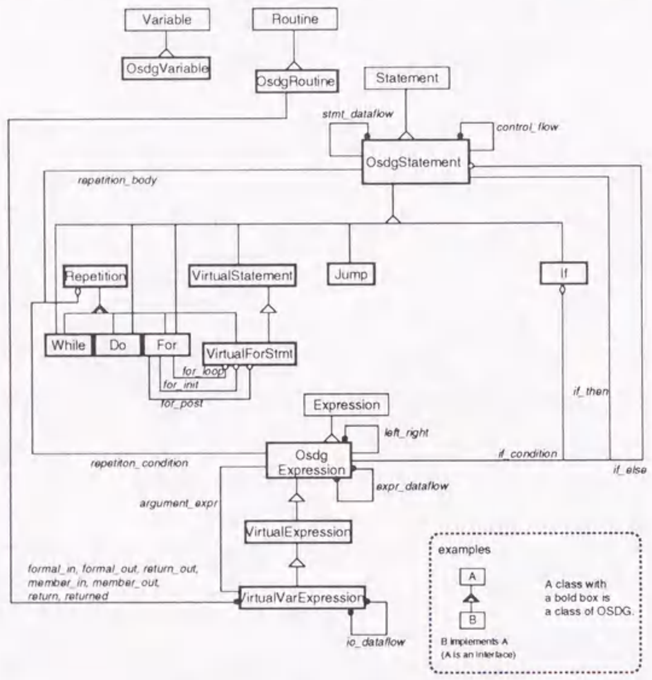


Figure 3.13: Class Diagram of OSDG

In our model, we specialize the *Statement* class into the *Repetition, If, Jump*, and *VirtualStatement* classes. The *Repetition* class shows `while`, `do`, and `for` statements. In this model, `for` statement is represented like a `while` statement (Figure 3.14). It enables treating two expressions of `for` statement,

E1 and *E3*, as normal *expression-statements*. On translation, new *virtual* statements, `E1;`, `while ...`, and `E3;`, are created. They are represented as the *VirtualForStatement* class in Figure 3.13. We show a program of these classes in Figure 3.15.

The *Jump* class represents `return`, `break`, and `continue` statements. We also show control flow graphs of some statements in Figure 3.16. Nodes of them represent statements and edges of them represent control flows.

```
S1;
for (E1; E2; E3)
    S2;
}
S3;
```

$\Longrightarrow$

```
S1;
E1;
while (E2) {
   S2;
   E3;
}
S3;
```

Figure 3.14: `for` Statement and Its Translation

The *VirtualVarExpression* class represents parameter nodes such as *formal_in* and *formal_out*.

We have not modeled *switch, exception, synchronized*, and *local class* statements yet. However, we analyzed the kinds of statements of 692 classes in JDK-1.1.6, and by its result, our model covers 94.7 percent of all statements (Table 3.1).

We have made tools that construct the OSDG, `MakeOSDG` and `LoadOSDG` with this model. `MakeOSDG` constructs graphs at three levels and saves them to the database. We have not implemented nodes and edges about a method call yet; `MakeOSDG` does not creat polymorphic choice nodes and method call, parameter passing, and polymorphic call edges. `LoadOSDG` loads data from the database, which are produced by `MakeOSDG`, and constructs a graph at a given level.

3.4.3 Performance

We measured the execution time and memory size of `MakeOSDG` and `LoadOSDG` at Sun Ultra1 OEM workstation (300 MHz UltraSPARC-II, 128MB RAM). A target program was `String` class of JDK-1.1.6 (1531 lines). The SDB of Japid provided 4245 objects and 6128 associations of the `String` class[1]. They included nodes and edges that were not used by the OSDG, such as *Type* objects and *has_type* associations. Table 3.2 shows the number of objects

[1] An object corresponds to a node and an association corresponds to an edge.

```
 1: public class OsdgStatement extends Statement {
 2:     protected Vector    nextStatements;
 3:     protected Vector    previousStatements;
 4:     ...
 5:
 6:     /** Returns statements which will execute next step. */
 7:     public Enumeration getNextStatements() {
 8:         ...
 9:     }
10:     /** Returns statements which may executed previous step . */
11:     public Enumeration getPreviousStatements() {
12:         ...
13:     }
14:     /**
15:       * Set nextStatements.
16:       * This method is overriden by While, For, and so on.
17:       */
18:     protected void setNextStatements() {
19:         ...
20:     }
21:     ...
22: }
23:
24: public interface Repetition {
25:     /** Returns loop body. */
26:     public OsdgStatement getBody();
27:     /** Returns loop condition. */
28:     public OsdgExpression getCondition();
29: }
30:
31: public class ForStatement extends OsdgStatement implements Repetition {
32:     /** for (init; condition; post) body */
33:     protected OsdgStatement body;
34:     protected OsdgExpression condition;
35:     protected OsdgVirtualForStatement init;
36:     protected OsdgVirtualForStatement loop;
37:         // means while (condition) {body; post;}
38:     protected OsdgVirtualForStatement post;
39:     ....
40:
41:     public OsdgStatement getBody() {
42:         return body;
43:     }
44:     public OsdgExpression getCondition() {
45:         return condition;
46:     }
47:
48:     /** Returns init expression as virtual statatement */
49:     public OsdgVirtualForStatement getInit() {
50:         return init;
51:     }
52:     /** Returns loop body as virtual statatement */
53:     OsdgVirtualForStatement getLoop() {
54:         return loop;
55:     }
56:     /** Returns post expression as virtual statatement */
57:     OsdgVirtualForStatement getPost() {
58:         return post;
59:     }
60:
61:     /** Set nextStatements. */
62:     protected void setNextStatements() {
63:         ...
64:     }
65:     ...
66: }
67:
68: public class VirtualForStatement extends OsdgVirtualStatement
69:         implements Repetition {
70:     protected expr;                     // used by init and post
71:
72:     protected OsdgStatement body;       // used by loop
73:     protected OsdgExpression condition; // used by loop
74:     ....
75:
76:     public OsdgStatement getBody() {
77:         return body;
78:     }
79:     public OsdgExpression getCondition() {
80:         return condition;
81:     }
82:     public OsdgExpression getExpression() {
83:         return expression;
84:     }
85: }
```

Figure 3.15: Program of Classes of OSDG

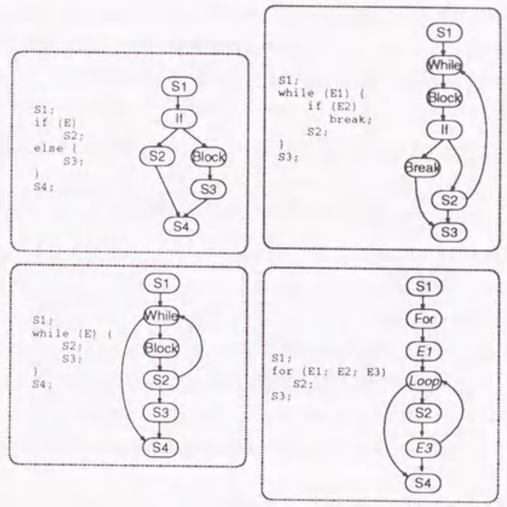


Figure 3.16: Control Flow

Table 3.1: Kinds of Statements in JDK-1.1.6

Kind of Statement (Modeled)	Number	Percentage
Empty	1246	3.0
Expression	10787	26.3
Block	11890	29.0
LocalVariable	4198	10.2
If	4854	11.8
While	322	0.8
Do	23	0.1
For	829	2.0
Break	507	1.2
Continue	81	0.2
Return	4119	10.0
Subtotal	38856	94.7
Kind of Statement (Unmodeled)	**Number**	**Percentage**
Switch	138	0.3
Exception Handler	851	2.1
Throw	899	2.2
Synchronized	292	0.7
Local Class	0	0.0
Subtotal	2180	5.3
Total	41036	100.0

related with the OSDG. `MakeOSDG` loaded objects and associations from SDB and created 4936 objects and 8338 associations [2] and saved them to the database in 16.9 seconds. It used 6.7 megabytes memory (Table 3.3).

Table 3.2: Number of J-model Objects of `String` Class

J-model Class	Number of Objects
Routine	61
Constructor	13
Method	48
Variable	214
Member Variable	4
Formal Parameter	91
Statement	538
Expression	1638

Table 3.3: Time and Memory for Loading SDB and Constructing OSDG

Type	Time [sec.]	Memory [MB]	Objects	Associations
Load SDB	9.6	4.3	4245	6128
`MakeOSDG`	16.9	6.7	4936	8338

We constructed the graphs at three levels using `LoadOSDG`. We show their results in Table 3.4. By them, we could get enough performance to use practically and we confirmed effectiveness of changing granularity of graph: a coarse grained graph is constructed in shorter time and less memory than a fine grained graph.

Table 3.4: Time and Memory for Loading OSDG

Granularity	Time [sec.]	Memory [MB]	Objects	Associations
Expression-grain	7.1	5.2	4018	2810
Statement-grain	4.6	4.2	2448	2054
Method-grain	3.3	2.9	1684	918

3.5 Application

In this section, we propose two applications using the OSDG, **object dependence analysis** and **slicing**. Traditional data dependence analyses are not accommodated to an object, which is a capsule of data and its operation. We propose object dependence analysis.

The second is the slice of classes. Program slicing has been proposed by Weiser[37]. We show algorithms of slicing an object-oriented program using reachability of the OSDG.

[2] `MakeOSDG` created 691 new objects and 2210 new associations.

3.5.1 Object Dependence Analysis

It is important to understand dependences between objects, because an object-oriented program performs computation by interactions between objects. However, traditional data dependence analyses are not accommodated to an object, whose state is changed not only by accessing a member variable but also by calling a method. In this section, we propose an object dependence. We get the lifetime of an object and a sequence of called methods on an object using object dependence analysis.

Definition 3.9 (Definition of Object) When a member variable of an object is defined, we call that the object is **defined**. □

Definition 3.10 (Use of Object) When an object is not defined and its member variable is used, we call that the object is **used**. □

Definition 3.11 (Object Reference Expression) When an expression E defines or uses an object and E is an elementary expression, we call E an **object reference expression**. □

Definition 3.12 (Object Flow) When object reference expressions E_1 and E_2 satisfy all of the following conditions, we say that there is an **object flow** for an object x from E_1 to E_2.

1. The expression E_1 defines the object x.
2. In the statement S_1, which includes the expression E_1, the object x is not defined after evaluating the expression E_1.
3. The expression E_2 uses the object x.
4. In the statement S_2, which includes the expression E_2, the object x is not defined before evaluating the expression E_2.
5. There is a control flow path from S_1 to S_2 and the variable x is not defined in statements included in its path. □

Definition 3.13 (Object Dependence) When there is an object flow from E_1 to E_2, we say that there is an **object dependence** from E_1 to E_2. □

The definition and the use of an object is classified into two kinds: a direct access to a member variable and an indirect access by a method call. Analyzing the definition and the use by a direct access is easy. However, in order to analyze the definition and the use by a method call, we must analyze side-effect by a method call.

The OSDG represents an operation on an object as an expression node labeled "ObjE". *member_in* represents a member variable used in a method and *member_out* represents a member variable defined in a method. We find the definition and the use of an object by a method call using the OSDG as follows:

> For an expression *O.m()*, which shows a method call m for an object O,
>
> - the object reference expression O is the definition, when there is a method parameter edge between *method entry* node of m and a *member_out* node.
> - the object reference expression O is the use, when there is no method parameter edge between *method entry* node of m and a *member_out* node and there is a method parameter edge between method entry node of m and a *member_in* node.

For example, in Figure 3.9, the object reference expression `obj` of `obj.h(0)` is the use of the object because the method `h` has the *member_in* node *h1* and it has no *member_out* node.

3.5.2 Slicing Object-Oriented Program

Program slicing is useful for applications such as debugging, program understanding, program testing, and so on. Program slicing is a technique for getting statements that may affect slicing *criterion* $< p, x >$; p is a program point and x is a variable used or defined at p[37][28].

We propose an algorithm for slicing using reachability of the OSDG. In the OSDG, slicing criterion is specified as a statement node and a variable reference expression node.

Algorithm 3.1 (Slice with Variable) By the following algorithm, we get a slice with respect to a statement node p and a variable reference expression node x.

0. Let S and E be sets such as $S = \phi, E = \phi$.
1. Find all expression and parameter nodes such that there are transitive data dependences from them to x. Add them to E.
2. For all $e \in E$, find the statement s that includes the expression e. Add s to S.
3. For all $s \in S$, find a statement s' such that there is a control flow from s' to s. Add s' to S.
4. The set S is the desired slice. □

We show an example of slicing in Figure 3.17. Its criterion is $< SC5, res >$. The statement *SC4*, `a1 = 0;`, is not selected for this slice.

We apply this algorithm to a graph having dynamic method calls. In this case, all methods that may be called are sliced. Figure 3.18 shows the slice with respect to $< St1, ret >$. Methods `A#m2(int)` and `C#m2(int)` that may be called from $St1$ are sliced.

In object-oriented programs, we often have to observe and set the state of an object by method calls because we cannot manipulate member variables directly. Thus, object-oriented programs substitute many variable references with method calls. Larsen and Harrold define program slicing for a method call[24]. Its criterion $< p, m >$ consists of a statement p and a method m called at p. We get this slice as union of slices of actual parameters.

Algorithm 3.2 (Slice with Method Call) We get a slice with respect to a statement node p and a method call expression node m by the following expression.

$$S = \bigcup_{p_a \in P_a} s(p, p_a)$$

S	:	a slice with respect to $< p, m >$
P_a	:	a set of actual parameter at m
$s(p, p_a)$	:	a slice with respect to $< p, p_a >$

□

We show an example of a slice with a method call in Figure 3.17. Its criterion is $< SC3, m2 >$. Desired slice consists of statements, *SA1*, *SA2*, *SC2*, and *SC3*.

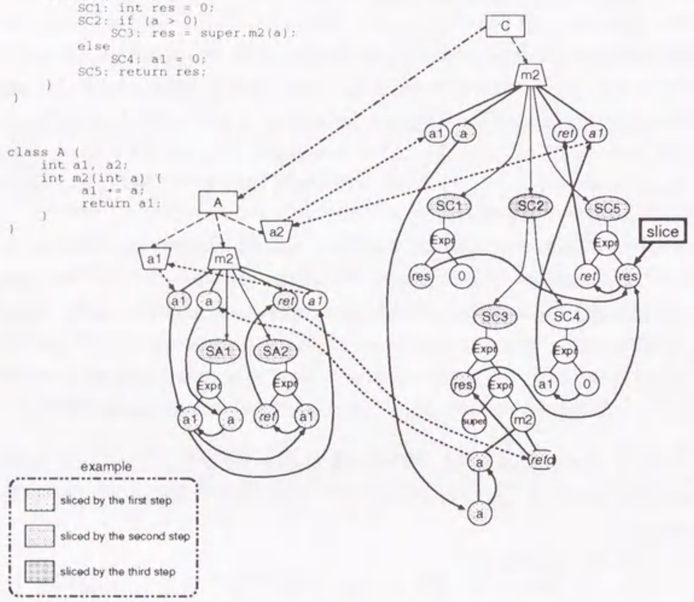


Figure 3.17: Slicing Using OSDG

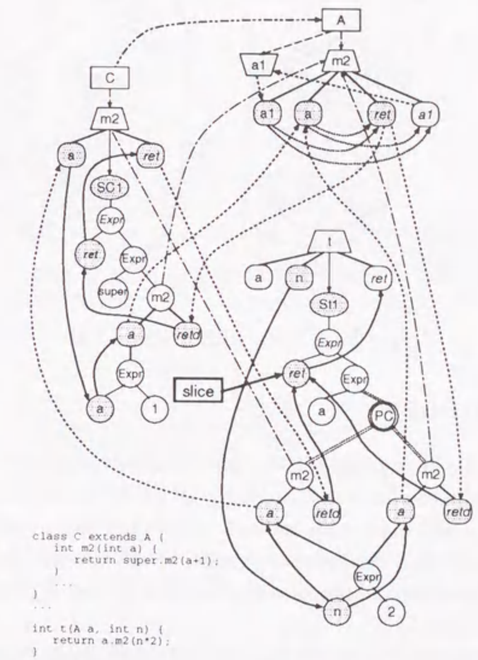


Figure 3.18: Slicing Using OSDG with Dynamic Method Call

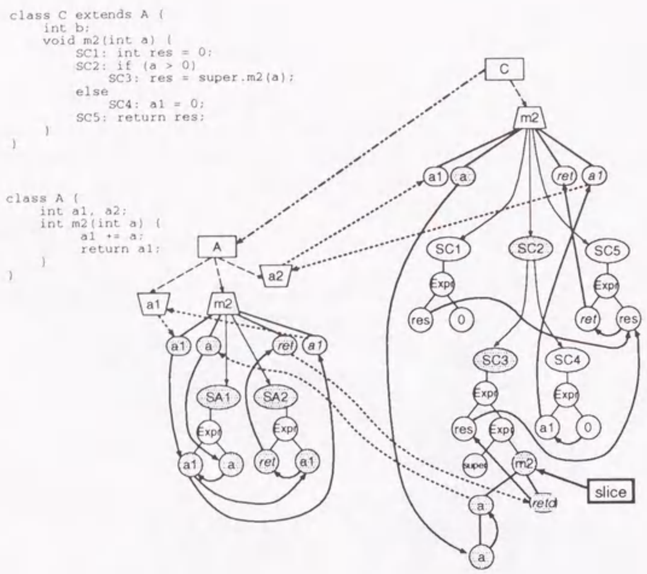


Figure 3.19: Slice with Method Call

3.6 Summary

In this chapter, we have proposed the OSDG, object-oriented system dependence graph, and a method for changing granularity of a graph. Because the OSDG is built at the expression-grain level, it expresses more precise information than traditional dependence graphs: it can express data dependence of method composition and one of a statement that has multiple occurrences of the same variable.

We have shown an implementation of the OSDG using Japid; fine grained information and the view definition mechanism provided by Japid are useful. We made some experiments. From their results, we could get enough performance to use practically and we confirmed effectiveness of changing granularity of a graph: a coarse grained graph is constructed in shorter time and less memory than a fine grained graph.

We have proposed application of the OSDG: object dependence analysis and algorithm for slicing object-oriented programs. In the next chapter, we use the OSDG for specializing object-oriented programs.

In future, we plan to implement `MakeOSDG` and `LoadOSDG` completely, i.e., to implement a part of method call, polymorphic choice nodes and method call, parameter passing, and polymorphic call edges.

Chapter 4

Specializing Object-Oriented Programs

In this chapter, to achieve the research goal (3), we propose three approaches to specialization of object-oriented programs by use of Japid and the OSDG. Not all object-oriented programs run efficiently in comparison with traditional procedural programs, because of dynamic method invocation, encapsulation, which prohibits optimizing methods between classes. Use of a specializer makes it possible to write a program elegantly and run it efficiently.

In this chapter, we propose three approaches to specializing object-oriented programs. First, we propose **class fusion** and **class reduction**, which are techniques merging closely associated classes into a single class. The second approach is **class slimming**, which removes unused methods and variables from classes of class libraries or from classes enlarged by class fusion and reduction. The last approach is static analysis of dynamic invocation, which replaces dynamic method invocation with static one. In this thesis, we describe four techniques for this analysis: the *unique name method*[6], *class hierarchy analysis*[9], *rapid type analysis*[3], and *flow-sensitive analysis*[3]. We also demonstrate their effectiveness through experiments.

4.1 Specialization Techniques

4.1.1 Intra-Class and Inter-Class Specialization

We classify specialization of object-oriented programs into two kinds: *intra-class* specialization and *inter-class* specialization. The former specializes statements and expressions in a method (e.g., loop unfolding), and method invocations between methods in the same class (e.g., method in-lining). Specialization techniques for procedural programs, which have already been proposed and recognized as very useful, can be applied as intra-class specialization because we can consider a class as a procedural program: we consider member variables as global variables, methods in the class as procedures and functions, and methods in other classes as library functions.

The second type, inter-class specialization is based on structure among some classes. In this section, we propose three approaches to inter-class specialization.

4.1.2 Class Fusion and Reduction

We propose a method for merging closely associated classes into a single class. We call this **class fusion**. Using class fusion, we can apply intra-class specialization to more classes and reduce the object instantiation overhead.

For example, we often use the *adapter* pattern to resolve interface mismatching, and an adapter class is defined only to match interfaces [17]. In this case, we can merge the adapter and adaptee classes (Figure 4.1).

Let us take another example. Assume that, in the design phase, we design a class A as an aggregation of classes B and C. In the implementation phase, the classes B and C are used only in the class A, and the class A has member variables that are instances of the classes B and C. In this case, the classes B and C can be merged into the class A (Figure 4.2).

We formalize class fusion as follows:

Definition 4.1 (Class Fusion) We can merge a class B into a class A if

1. in an entire program, the class B is used only in the class A, and
2. the class A has a member variable that is an instance of the class B.

We call this merging **class fusion**. An entire program means a program that includes all required classes and methods. □

The class B is merged into the class A as follows (Figure 4.3):

1. A member variable of the class A that is an instance of the class B is removed.
2. Member variables and methods of the class B are added to the class A. When their names conflict with members and methods of the class A, they are substituted; names of declaration and reference are replaced. For example, `print()` of the class `Adaptee` is replaced with

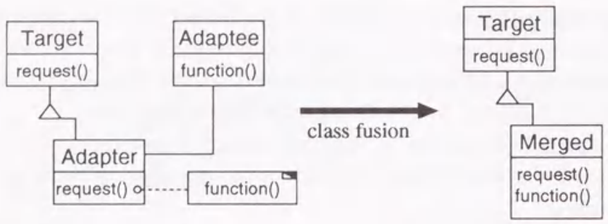


Figure 4.1: Class Fusion (Adapter pattern)

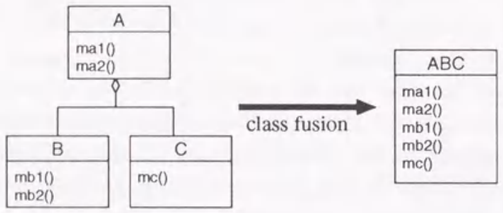


Figure 4.2: Class Fusion (Aggregation of classes)

`printAdaptee()`. Method invocations and member references on an instance of the class B are replaced with direct accesses. For example, `adp.function()` is replaced with `function()`.

3. Constructors of the class B are translated into methods and added to the class A. For example, the return type `void` is added to the constructor declaration (`void Adaptee() {...}`), and a constructor call such as `this()` is replaced with `Adaptee()`. An object creation expression is replaced with a method invocation. For example, `new Adaptee()` is replaced with `Adaptee()`.

Class Reduction

We extend the idea of class fusion to inheritance classes. We merge a class and its subclasses into a single class. We call this **class reduction**.

Definition 4.2 (Class Reduction) We can merge classes $A_1, A_2, \ldots, A_{n-1}$ into a class A_n if


```
 1: class Adaptee {
 2:    // member
 3:    int member;
 4:
 5:    // constructors
 6:    Adaptee(int m) {...}
 7:    Adaptee() {this(0);}
 8:
 9:    // methods
10:    void function() {...}
11:    void print() {adp.print(); ...}
12:    ...
13: }
14:
15: class Adapter {
16:    // member
17:    Adaptee adp;
18:
19:    // constructor
20:    Adapter() {adp = new Adaptee();}
21:    }
22:
23:    // methods
24:    void request() {adp.function();}
25:    void print() {...}
26:    ...
27: }
28:
29:
30: /* === class fusion ================ */
31: class Adapter { // merged Adaptee
32:    // constructor
33:    Adapter() {Adaptee();}
34:
35:    // methods
36:    void request() {function();}
37:    void print() {printAdaptee(); ...}
38:    ...
39:
40:    // member of Adaptee
41:    int member;
42:
43:    // constructors of Adaptee
44:    void Adaptee(int m) {...}
45:    void Adaptee() {Adaptee(0);}
46:
47:    // methods of Adaptee
48:    void function() {...}
49:    void printAdaptee() {...}
50:    ...
51: }
```

Figure 4.3: Class Fusion (Program of Adapter Pattern)

1. there are inheritance associations $A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n$.
   "*Base* $\rightarrow$ *Derived*" means that the class *Base* is a superclass of the class *Derived*.

2. In an entire program, classes $A_1, A_2, \ldots, A_{n-1}$ have no instances, and

3. there is no subclass of the class A_i $(1 \leq i \leq n-1)$ other than A_j $(j > i)$.

We call this merging **class reduction**. □

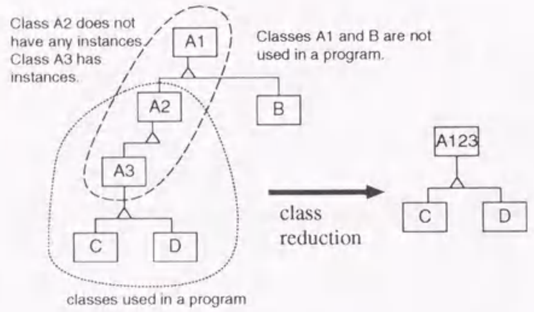


Figure 4.4: Class Reduction

4.1.3 Class Slimming

Class slimming is a technique for removing unused methods and variables. We define class slimming as follows.

Definition 4.3 (Class Slimming) By **class slimming** with respect to method **m**, we obtain classes that have methods, constructors, and member variables that might be directly or indirectly invoked and referred to from **m**. □

Class slimming is useful for removing unused methods and variables from general-purpose classes in a class library or from classes enlarged by class fusion and reduction.

Classes obtained by class slimming with respect to the starting-up method includes only classes, constructors, methods, and variables that are necessary for a program. This is an entire program that is used for class fusion and reduction, and for static analysis of dynamic invocation (see the following section).

We perform class slimming by using a call graph from method *m*. Constructing an accurate call graph of an object-oriented program is difficult because of dynamic invocations. Grove et al show an algorithm for constructing a call graph in [7].

4.1.4 Static Analysis of Dynamic Invocation

Generally, in object-oriented programs, an invoked method is bound dynamically at run-time. While this feature enables a program to be flexible and reusable, it causes overhead at run-time and difficulty of inter-method analysis.

In this section, we describe four techniques of static analysis of dynamic invocation, the *unique name method*[6], *class hierarchy analysis*[9], *rapid type analysis*[3], and *flow-sensitive analysis*[3]. We use a small program in Figure 4.5 in order to illustrate the difference among these analysis methods. There are three dynamic invocations in `main(String args[])`.

A compiler often uses these analyses for optimizing codes. In this thesis, we use them for specializing; a specializer rewrites a program: for example, from `void foo() {...}` to `final void foo {...}`[1].

Unique Name

When a method has a unique name (really a unique signature) in a program, invocations of it are bound statically. This technique is called unique name method. In Figure 4.5, unique name method resolves only the first invocation, which produces `result1`, because there are two methods that have a name `foo` and take an integer parameter (`A#foo(int)`, `B#foo(int)`).

Class Hierarchy Analysis

Class hierarchy analysis uses combination of the statically declared type of an object and the class hierarchy of an entire program. The statically declared type means the type in an object declaration. For example, in an object declaration "`String str;`", the statically declared type of `str` is `String`.

[1] In Java, a method declared with the `final` keyword cannot be overridden by subclasses and a compiler often optimizes it.

```
 1: class A {
 2:     int foo() { return 1; }
 3: }
 4:
 5: class B extends A {
 6:     int foo() { return 2; }
 7:     int foo(int i) { return i+1; }
 8: }
 9:
10: class Test {
11:     public static void main(String args[]) {
12:         B b = new B();
13:         int result1 = b.foo(2);  // UN, CHA, RTA
14:         int result2 = b.foo();   // CHA, RTA
15:         A a = b;
16:         int result3 = a.foo();   // RTA
17:     }
18: }
```

Figure 4.5: Program Including Three Dynamic Method Invocations

Note that it is not represented the real type of `str`, because `str` may be an object of a subclass of `String`. This means that an invoked method on `str` is a method of the class `String` or of subclasses of `String`. By combination of this information and the class hierarchy, we can resolve methods of a class that has no subclass.

In Figure 4.5, the second invocation, which produces `result2`, is resolved by class hierarchy analysis, because the statically declared type of `b` is `B` and the class `B` has no subclass in the entire program. The first invocation is also resolved by class hierarchy analysis. However, the third invocation, which produces `result3`, is not resolved, because the statically declared type of `a` is `A` and the class `A` has a subclass `B`.

Class hierarchy analysis resolves more method invocations than unique name method, because it eliminates methods that have the same signature in unrelated classes from possible targets.

Class hierarchy analysis needs an entire program, because if another module defines a class `C` that is a subclass of the class `B` and overrides a method `foo()`, then the invocation cannot be resolved.

Rapid Type Analysis

Rapid type analysis searches an entire program for really instantiated objects. Only methods of classes that have their instances are invoked.

In Figure 4.5, it resolves the third invocation by following steps.

1. The statically declared type of `a` is `A`. This means methods that may be invoked are `A#foo(int)` and `B#foo(int)`.

2. The entire program shows that no object of the class `A` is instantiated and an object of the class `B` is instantiated. `A#foo(int)` is eliminated from possible targets and only `B#foo(int)` is leaved.

Rapid type analysis resolves more methods than class hierarchy analysis, because it eliminates methods of a class that has no instance from possible targets.

Flow-Sensitive Analysis

In the following program, rapid type analysis cannot resolve the invocation, because both objects of A and B are instantiated. In order to resolve it, data flow analysis is needed. We call such an analysis method flow-sensitive analysis.

```
A a = new B();
a = new A();
result = a.foo();
```

For method invocations on objects such as `obj.method()`, flow-sensitive analysis recognizes which of the statements create an object and data flows from it. This analysis can resolve only invocations on objects, while other three analyses analyze all method invocations. Flow-sensitive analysis is classified into some levels: local flow-sensitive analysis, inter-method flow-sensitive analysis, and so on. Local flow-sensitive analysis recognizes data flows between statements in only one method; it can resolve method invocations on objects created in the same method. Inter-method flow-sensitive analysis analyzes data flows between methods; it can resolve invocations on an object that is a parameter of a method and created in other methods. The OSDG proposed in Chapter 3 is useful for flow-sensitive analysis.

Class hierarchy and rapid type analyses need a call graph from the starting-up method. However, in a program with graphical user interfaces (GUI), some methods invoked by a user's actions such as clicking a button may not be included in a call graph; these analyses cannot be applied to this kind of program. On the other hand, flow-sensitive analysis can be applied to it.

Local flow-sensitive analysis is not affected by a target program: resolved invocations in a class are identical for any program. However, other analyses are affected by a target program: according to target programs, resolved invocations are different.

4.2 Experiments

In this section, we describe some experiments that we carried out to demonstrate the effectiveness of our approach. The first experiment is related to class fusion and reduction: we show applicability of class fusion and reduction, and the speedup obtainable by class fusion and reduction. The second experiment is related to class slimming: we show ratios of used methods, constructors, and member variables to declared ones and an example of applying class slimming to a simple program. The last experiment shows applicability of rapid type and local flow-sensitive analyses. We made some tools for experiments by using Japid and the OSDG.

We show sizes of target programs of experiments in Table 4.1. The numbers of files, classes, etc., do not include class libraries of JDK. *Bubble Sorting* sorts integers by means of the bubble sorting algorithm. In our experiments, we specialized the essential parts of a sorting program, which compare and swap objects, and not the parts that read integers from a file (see Section 4.2.1). To clarify the effect of specializing parts that compare and swap objects and to hide the overhead of reading a file, we selected the bubble sorting algorithm. *JLex* is a lexical analyzer written in Java. *JDK Demos* include five programs. *JDK Demos*, *RPN (Reverse Polish Notation) Calculator*, and *Calendar* programs have graphical user interfaces. *JDK-1.1.6* is the standard class libraries provided by Sun: *java.lang*, *java.io*, and so on. It does not include `main(String args[])` method, which is a starting-up method of Java.

4.2.1 Class Fusion and Reduction

We carried out two experiments about class fusion and reduction: their applicability and their speedup.

The first experiment shows the extent to which class fusion and reduction can be applied to practical programs. We show its result in Table 4.2. *Bubble Sorting* used forty classes, which included classes of JDK's class libraries. The classes had twenty-four inheritance associations. We found seven classes for class fusion and seven classes for class reduction. *Grep*, *Tar*, etc., were carried out in the same way.

Table 4.1: Program Size

Program	Files	Classes	Intf.	Cons.	Methods	Lines
Bubble Sorting	3	3	0	5	12	163
Grep	18	18	0	27	103	1991
Tar	7	6	1	16	63	1903
JLex	1	20	0	18	111	7351
JDK Demos (5 programs)	6	24	0	14	65	1481
RPN Calculator	11	11	0	16	102	1915
Calendar	3	3	0	3	19	415
JDK-1.1.6	663	578	116	772	5001	148351

Intf.: Interfaces, Cons: Constructors

We confirmed that class fusion and reduction can be applied to practical programs.

Table 4.2: Applicability of Class Fusion and Reduction

Program	Number of Classes			
	Total	Inh.	Fusion	Red.
Bubble Sorting	40	24	7	7
Grep	52	24	4	6
Tar	56	35	4	8
JLex	85	28	2	3
JDK Demos	113	71	20	17
RPN Calculator	69	38	7	11

Inh.: Inheritance, Red.: Reduction

The second experiment shows the speedup that can be obtained through the use of class fusion and reduction. We selected *Bubble Sorting* and *Grep*.

We show a program of Bubble Sorting in Figures 4.6 and 4.7. A program of Figure 4.6 is the unspecialized original program, and one of Figure 4.7 is a program specialized by class reduction. The class `ArraySorter` in Figure 4.6 sorts an array of integers by means of the bubble sorting algorithm. It has an abstract method `int compare(int, int)`. It can sort an array in ascending or descending order using a comparison method supplied by a subclass. The class `SortIntArray` overrides a comparison method for ascending order. The class `SortIntFile` is a driver for reading and sorting integers.

By class reduction, the class `ArraySorter` and the class `SortIntArray` were merged into the class `SortIntArray2` in Figure 4.7. This version has

two classes, `SortIntArray2` and `SortIntFile2`, which is a minor changed version of the class `SortIntFile` for using the class `SortIntArray2`.

We compiled merged classes with an optimization option (`-O`), which applies intra-class specialization to them. We show the execution time of them in Table 4.3 (Bubble Sorting) and Table 4.4 (Grep). "Hand specialized" in Table 4.3 means a program specialized by hand: `SortIntArray2` and `SortIntFile2` were merged into a single class and redundant codes were deleted.

Table 4.3: Speedup (Bubble Sorting)

Case	Time [sec.]	Speedup
Not merged	52.5	1.00
1 reduction	42.4	1.23
hand specialized	42.4	1.23

Table 4.4: Speedup (Grep)

Case	Time [sec.]	Speedup
Not merged	55.3	1.00
1 fusion	54.9	1.00
1 fusion and 1 reduction	53.8	1.02

In the case of Bubble Sorting, an unspecialized program sorted five thousand integers in 52.5 seconds. The program to which class reduction was applied completed sorting in 42.4 seconds. It ran 23 percent faster than the unspecialized program.

In the case of Grep, however, class fusion and reduction did not result in any speedup.

In order to clarify the difference between Bubble Sorting and Grep, we checked byte-codes of them (Tables 4.5 and 4.6). In the case of Bubble Sorting, the number of method invocations, particularly dynamic method invocations, was reduced by class reduction. In a specialized program, a comparison method `int compare(int, int)` was recognized as a static invocation and it was in-lined. It resulted in the speedup.

In the case of grep, however, the number of method invocations was hardly reduced. There was an increase in the number of dynamic method invocations, due to the addition of constructor calls of the unspecialized program.


```
1: abstract class ArraySorter {
2:     int array[];
3:
4:     void setArray(int[] array) {
5:         this.array = array;
6:     }
7:     int[] getArray() {}
8:         return array;
9:     }
10:
11:    abstract int compare(int a, int b);
12:
13:    /** Sort array by bubble sort algorithm  */
14:    public void bsort() {
15:        ...
16:    }
17:
18:    /** Swap array[i] and array[j] */
19:    protected void swap(int i, int j) {
20:        ...
21:    }
22: }
23:
24: ////////////////////////////////////////////////
25: final class SortIntArray extends ArraySorter {
26:    /** Compares two numbers for ascending order */
27:    int compare(int a, int b) {
28:        if (a < b)      return -1;
29:        else if (a > b) return 1;
30:        else            return 0;
31:    }
32: }
33:
34:
35: ////////////////////////////////////////////////
36: final class SortIntFile {
37:    int array[];
38:    ArraySorter sorter;
39:
40:    void readFile(File file) {
41:        // read file and set array
42:    }
43:
44:    void setSorter(ArraySorter sorter) {
45:        this.sorter = sorter;
46:    }
47:    void sort() {
48:        sorter.setArray(array);
49:        sorter.bsort();
50:    }
51:
52:    public static void main(String args[]) {
53:        SortIntFile fileSorter = new SortIntFile(args[0]);
54:        fileSorter.readFile();
55:        fileSorter.setSorter(new SortIntArray());
56:        fileSorter.sort();
57:    }
58: }
```

Figure 4.6: Unspecialized Bubble Sorting Program

```
 1: final class SortIntArray2 {
 2:     int array[];
 3:
 4:     void setArray(int[] array) {
 5:         this.array = array;
 6:     }
 7:     int[] getArray() {}
 8:         return array;
 9:     }
10:
11:     int compare(int a, int b) {
12:         if (a < b)      return -1;
13:         else if (a > b) return 1;
14:         else            return 0;
15:     }
16:
17:     /** Sort array by bubble sort algorithm  */
18:     public void bsort() {
19:         ...
20:     }
21:
22:     /** Swap array[i] and array[j] */
23:     protected void swap(int i, int j) {
24:         ...
25:     }
26: }
27:
28: ////////////////////////////////////////////////
29: final class SortIntFile2 {
30:     int array[];
31:     SortIntArray2 sorter;
32:
33:     void readFile(File file) {
34:         // read file and set array
35:     }
36:
37:     void setSorter(SortIntArray2 sorter) {
38:         this.sorter = sorter;
39:     }
40:     void sort() {
41:         sorter.setArray(array);
42:         sorter.bsort();
43:     }
44:
45:     public static void main(String args[]) {
46:         SortIntFile fileSorter = new SortIntFile(args[0]);
47:         fileSorter.readFile();
48:         fileSorter.setSorter(new SortIntArray2());
49:         fileSorter.sort();
50:     }
51: }
```

Figure 4.7: Bubble Sorting Program Applied Class Reduction

Table 4.5: Analysis of Byte-code (Bubble Sorting)

Case	Byte-code [lines]	Method Invocations Dynamic / Total
Not merged	243	25/44
1 reduction	251	22/39
hand specialized	208	20/34

Table 4.6: Analysis of Byte-code (Grep)

Case	Byte-code [lines]	Method Invocations Dynamic / Total
Not merged	1422	135/243
1 fusion	1421	136/242
1 fusion and 1 reduction	1419	137/241

4.2.2 Class Slimming

We carried out two experiments about class slimming. First, we show ratios of used methods, constructors, and member variables to declared ones in practical programs. Second, we show an example of applying class slimming to a simple program.

First, we show ratios of invoked methods to declared ones, of invoked constructors to declared ones, and of referred member variables to declared ones in practical programs (Table 4.7).

In this experiment, we found methods and constructors that are included in the call graph from the `main(String args[])` method as follows.

Algorithm 4.1 We find methods included in a call graph from a method m_0 as following ways.

0. Let M and C be sets such as $M = \phi, C = \phi$; M shows methods and constructors included in a call graph and C shows classes defining methods in M.

1. First, we construct a call graph from a given method (m_0) optimistically: we suppose that every method invocation is bound statically. We add methods and constructors in a call graph to M and add classes defining them to C.

2. Let M' be a set such as $M' = \phi$. We search classes in C for methods that overrides a method in M. We add them to M'.

3. If M' is empty, M is the desired set, which includes all methods and constructors that may be invoked. Otherwise, we construct a call graph from each method in M' optimistically. We add invoked methods and constructors to M and classes defining them to C. Go back to 2. □

This algorithm is analogous to class hierarchy analysis. We also counted member variables referred to by the methods and constructors in the call graph. We do not apply this algorithm to programs with GUI because they have methods that are not invoked from the `main` methods.

The results show that programs include many unused methods, constructors, and member variables (Table 4.7).

Table 4.7: Ratios of Used Methods, Constructors and Members to Declared Ones

Program	Methods	Constructors	Member Variables
Bubble Sorting	72/477 (15.0%)	24/ 77 (31.1%)	46/122 (37.7%)
Grep	152/594 (25.5%)	42/ 95 (44.2%)	142/214 (66.3%)
Tar	177/682 (25.9%)	52/117 (44.4%)	136/247 (55.0%)
JLex	193/721 (26.7%)	53/122 (43.4%)	252/565 (44.6%)

Each cell shows "the number of used / the number of declaration(percentage)". *Used* means invoked methods, invoked constructors, or member variables referred to.

Second, we show an example of applying class slimming. We make a simple queue class `Queue` that has enqueue and dequeue methods (Figure 4.8). It is designed as an adapter for the class `Vector`, which is an array of any objects and grows in size as necessary. The `Vector` class has three constructors and twenty-five methods and four member variables. The `Queue` class invokes directly one constructor[2] and three methods[3] of `Vector` and refers to no member variable directly. By class slimming with the method `main(String args[])` of the `Queue` class, which is a driver method for `Queue`, we obtain three constructors, four methods, and three member variables of `Vector`; two constructors[4] and one method [5] are invoked indirectly, and three member variables[6] are referred to indirectly. We can delete twenty-one redundant methods and one redundant member variable of `Vector` by class slimming.

[2] `Vector()`
[3] `addObject(Object)`, `elementAt(int)`, and `size()`
[4] `Vector(int)`, `Vector(int, int)`
[5] `ensureCapacityHelper(int)`
[6] `Object elementData[]`, `int elementCount`, and `int capacityIncrement`

In this case, we can apply class fusion to the `Queue` and the `Vector` classes. We can delete redundant methods and member variables from a merged class and obtain a queue class specialized as a program component (Figure 4.9). That is to say, we get a specialized program component from general-purpose programs, owing to class fusion and class slimming.

4.2.3 Static Analysis of Dynamic Invocation

We carried out two experiments about static analysis of dynamic invocations: we applied rapid type and local flow-sensitive analyses to practical programs and applied local flow-sensitive analysis to class libraries and programs with graphical user interfaces.

First, we show how many dynamic invocations of practical programs can be resolved by rapid type and local flow-sensitive analyses (Table 4.8). We constructed call graphs from the `main` methods by Algorithm 4.1 and counted method invocations included in them.

Because *JDK Demos* have graphical user interfaces, the invocations resolved by rapid type analysis may not be accurate: invocations that cannot be resolved by rapid type analysis may be included. We show it for reference.

The result shows that many dynamic invocations can be bound statically by rapid type analysis. On the other hand, local flow-sensitive analysis resolves few invocations and resolved invocations can also be resolved by rapid type analysis. In JDK Demos, however, it resolves invocations that cannot be resolved by rapid type analysis. These programs sometimes have subclasses of GUI components such as Figure 4.10. In the figure, rapid type analysis cannot resolve `p.add()` in line 4 because both of the class `Panel` and its subclass `PanelA` have their instances.

Second, we applied local flow-sensitive analysis to *JDK-1.1.6* and programs with graphical user interfaces (Table 4.9). We counted invocations on objects included in each program. The ratios of resolved invocations to all invocations on objects of programs with GUI are higher than the ratio of JDK-1.1.6.

4.3 Summary

In this chapter, we have presented three techniques to specialize object-oriented programs: class fusion and reduction, class slimming, and static analysis of dynamic invocation. We have shown their effectiveness by some experiments.

```
 1: import java.util.Vector;
 2:
 3: public class Queue {
 4:     private Vector v;
 5:     private int index;
 6:
 7:     public Queue() {
 8:         v = new Vector();
 9:         index = 0;
10:     }
11:
12:     public void enqueue(Object obj) {
13:         v.addElement(obj);
14:     }
15:
16:     public Object dequeue() {
17:         if (index >= v.size()) {
18:             return null;
19:         }
20:         return v.elementAt(index++);
21:     }
22:
23:     /**
24:      * A driver routine for testing the Queue class
25:      *   Add an argument to queue.
26:      *   If an argument equals "-", dequeue one element.
27:      */
28:     public static void main(String args[]) {
29:         Queue q = new Queue();
30:         for (int i = 0; i < args.length; i++) {
31:             if (args[i].equals("-")) {
32:                 System.out.println(q.dequeue());
33:             }
34:             else {
35:                 q.enqueue(args[i]);
36:             }
37:         }
38:     }
39: }
```

Figure 4.8: Queue Class Using Vector


```
 1: /** Merge the Vector class to the Queue class */
 2: public class ComponentQueue {
 3:     private int index;
 4:
 5:     public ComponentQueue() {
 6:         Vector();
 7:         index = 0;
 8:     }
 9:
10:     public void enqueue(Object obj) {
11:         addElement(obj);
12:     }
13:
14:     public Object dequeue() {
15:         if (index >= size()) {
16:             return null;
17:         }
18:         return elementAt(index++);
19:     }
20:
21:     /* === merged from the Vector class ==================== */
22:     protected Object elementData[];
23:     protected int elementCount;
24:     protected int capacityIncrement;
25:
26:     public Vector(int initialCapacity, int capacityIncrement) {
27:         ...
28:     }
29:     public Vector(int initialCapacity) {
30:         Vector(initialCapacity, 0);
31:     }
32:     public SimpleVector() {
33:         Vector(10);
34:     }
35:
36:     private void ensureCapacityHelper(int minCapacity) {
37:         ...
38:     }
39:     public final int size() {
40:         return elementCount;
41:     }
42:     public final synchronized Object elementAt(int index) {
43:         ...
44:     }
45:     public final synchronized void addElement(Object obj) {
46:         ...
47:     }
48:
49: }
```

Figure 4.9: Specialized Queue Class

Table 4.8: Number of Dynamic Invocations Resolved by RTA and LFS

Program	Number of Invocations				
	RTA	LFS	only LFS	on Object	All
Bubble Sorting	27 (31.8%)	4 (4.7%)	0 (0.0%)	42	85
Grep	225 (76.3%)	3 (1.0%)	0 (0.0%)	166	295
Tar	202 (64.5%)	14 (4.5%)	0 (0.0%)	137	313
JLex	273 (57.8%)	34 (7.2%)	0 (0.0%)	134	472
JDK Demos	354 (28.1%)	76 (6.0%)	20 (1.6%)	609	1260

RTA: invocations resolved by rapid type analysis
LFS: invocations resolved by local flow-sensitive analysis
only LFS: invocations resolved only by local-flow sensitive analysis (not resolved by rapid type analysis)
on Object: invocations on objects (e.g., `obj.method()`)
All: all dynamic invocations
A number in parenthesis shows the percentage of resolved invocations to all invocations.

```
1: // The class Panel is a container of GUI components.
2: // The class PanelA is a subclass of the class Panel.
3: Panel p = new Panel();
4: p.add(new Label("Normal Panel")); // not resolved by RTA
5: PanelA pa = new PanelA();
6: pa.add(new Label("Specific Panel"));
```

Figure 4.10: GUI Program

Table 4.9: Number of Dynamic Invocations Resolved by LFS

Program	Number of Invocations		
	LFS	on Object	All
JDK-1.1.6	294 (8.0%)	3696	7859
JDK Demos	77 (29.2%)	259	378
RPN Calculator	61 (17.1%)	357	454
Calendar	10 (27.0%)	37	70

LFS: invocations resolved by local flow-sensitive analysis
A number in parenthesis shows the percentage of resolved invocations to all invocations on objects
on Object: invocations on objects (e.g., `obj.method()`)
All: all dynamic invocations

Class fusion and reduction are applicable to practical programs, and a specialized program ran 23 percent faster than unspecialized one. Owing to class fusion and reduction, a dynamic invocation was replaced with static one and it was in-lined. But in another case, class fusion and reduction did not result in any speedup. We must find a method estimating the effect of class fusion and reduction before they are applied.

We can delete redundant methods and member variables by class slimming. We can obtain a specialized program component from general-purpose programs by use of class fusion and class slimming.

Rapid type analysis resolves many dynamic invocations of practical programs and local flow-sensitive analysis is useful for programs with graphical user interfaces.

In future, we plan to find a method estimating the effect of class fusion and reduction, and to implement a specializer with Japid completely.

Chapter 5

Conclusion

In this thesis, we have proposed a CASE tool platform for Java, Japid, and its applications: the object-oriented system dependence graph and specialization for object-oriented programs.

5.1 Achievements of Research Goals

We show achievements of our three research goals established in Section 1.2.

1. Letting developers be free from laborious work

We have achieved this goal. In Chapter 2, we showed a design and implementation of Japid. It provides fine grained information of analyzed programs and supports the view definition and program change mechanisms. We made some tools using Japid and we confirmed following points.

- Japid provides enough information and API for developing CASE tools. We could write them in a few codes.
- The view definition mechanism is useful for CASE tools. We implemented the writer, the object-oriented system dependence graph, etc., using user defined views.
- We could get acceptable performance.

2. Providing basic techniques

We have achieved this goal. In Chapter 3, we proposed the OSDG, a dependence graph for an object-oriented program. Many dependence graphs have already been proposed, but some problems are left; they cannot show a

data dependence of method composition and one of a statement having some occurrences of the same variable.

The OSDG is built at the *expression-grain* level. It can express a data dependence more precisely than graphs proposed previously. It solves problems mentioned above. We also proposed a method for decreasing the amount of data by changing granularity of a graph.

Dependence analysis is one of the most important techniques for software development. We showed algorithms for program slicing and specialization techniques using OSDG. We plan to implement other basic techniques such as an interpreter.

3. Developing new techniques

We have achieved this goal. In Chapter 3, we proposed the OSDG and object dependence analysis, which are new techniques related to object-oriented features. In Chapter 4, we also proposed three approaches to specialization of object-oriented programs: class fusion and reduction, class slimming, and static analysis of dynamic method invocation. We carried out some experiments using Japid and confirmed their effectiveness.

Japid is useful for developing these new techniques. We hope that many other techniques are developed using Japid.

5.2 Future Work

In this thesis, we focus on software development for lower phases such as an implementation phase. Conceptually, an object-oriented paradigm can offer uniformity throughout each phase of development[34]. We plan to support upper phases of development. For example, the object modeling technique (OMT) is used for object-oriented analysis and design[30]. The OMT methodology produces some documents such as object diagrams and state transition diagrams. If a CASE tool platform manages these documents, we can check consistency among analysis, design, and implementation phases. We can also translate specification documents to templates of programs or vice versa. This management is future work.

In this thesis, we model syntax and semantic of Java and our model focuses on an *identifier* such as a class, a method, and a variable. It represents static parts of object-oriented features such as encapsulation and inheritance. Other important object-oriented features are dynamic parts of object-oriented programs such as polymorphism, instantiation, and message-passing. They are represented as interactions between objects. We need to

execute a program actually for getting precise information of them. However, to some extent, we can get these dynamic information statically; for example, we can narrow possible targets of a dynamic method invocation (see Section 4.1.4). We plan to model interactions between objects using object dependence (see Section 3.5.1); we can create an event trace diagram, which shows the lifetime of an object and messages sent and received by the object. We also plan to make an interpreter for getting more precise dynamic information.

We plan to integrate CASE tools. Three types of CASE tool integration are proposed: data integration, display integration, and control integration[36]. Data integration is sharing and restructuring data of a repository. Display integration is sharing graphical user interfaces and defining common protocol for interacting users. Control integration is that a CASE tool communicates and cooperates with other CASE tools. Japid supports data integration by the access library and user defined views, but other integrations are not supported. We plan to support them. To some extent, display integration is done using the class library such as Java AWT (abstract window toolkit) and Swing, which are GUI component kits[39][33]. We plan to provide GUI components specialized for Japid using AWT or Swing: for example, a *text area* component displaying a source program. We also plan to define a framework for control integration using JavaBeans, which is a component architecture for Java[12].

Acknowledgement

First of all, I would like to thank my advisor Professor Kiyoshi AGUSA and my co-advisor Associate Professor Shinichirou YAMAMOTO currently of Aichi Prefectural University. Their assistance and guidance have made the completion of this dissertation possible. And I would like to thank the examiners Professor Toyohide WATANABE and Professor Toshiki SAKABE for their useful comments on this dissertation. I also thank Research Associate Atsushi YOSHIDA at Toyohashi University of Technology for his suggestion.

I thank all members of Agusa laboratory of Nagoya University for their discussion. Especially, I thank Takaaki SUZUKI and Hironori SUZUKI for developing CASE tools in Section 2.3.2.

Finally, I would like to thank my family for their support through the years.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1986.

[2] Ken Arnold and James Gosling. *The Java Language Specification, 2nd Edition.* Addison-Wesley, 1998.

[3] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the ACM 1996 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, October 1996.

[4] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System.* Morgan Kaufman, August 1991.

[5] Alan W. Brown, David J. Carney, Edwin J. Morris, Dennis B. Smith, and Paul F. Zarrella. *Principles of CASE Tool Integration.* Oxford University Press, Inc., 1994.

[6] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *21st ACM Symposium on Principles of Programming Languages (POPL)*, pages 397–408, January 1994.

[7] Craig Chambers, David Grove, Greg DeFouw, and Jeffrey Dean. Call Graph Construction in Object-Oriented Languages. In *Proceedings of the ACM 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, October 1997.

[8] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, Reading, Mass., fifth edition, 1991.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of 1995 European Conference on Object-Oriented Programming (ECOOP)*, pages 77–101. Springer-Verlag, August 1995.

[10] Premkumar T. Devanbu. GENOA: a customizable language- and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307–317, 1992.

[11] Premkumar T. Devanbu, David S. Rosenblum, and Alexander L. Wolf. Automated construction of testing and analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 241–250, 1998.

[12] Robert Englander. *Developing Java Beans.* O'Reilly & Associates, Inc., 1997.

[13] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use In Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, Jul 1987.

[14] David Flanagan. *Java in a Nutshell, 2nd Edition.* O'Reilly & Associates, Inc., 1997.

[15] N. Fukuyasu, S. Yamamoto, and K. Agusa. Consistent Changes of Source Programs based on the Fine Grained Software Repository (in Japanese). In *the 14th Conference Proceedings of JSSST*, pages 601–604, Oct 1997.

[16] N. Fukuyasu, S. Yamamoto, and K. Agusa. CASE Tool Platform Sapid based on a Fine Grained Repository (in Japanese). *Transaction of IPSJ*, 39(6):1990–1998, Jun 1998.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[18] Judith Grass and Y. F. Chen. The C++ Information Abstractor. In *The Second USENIX C++ Conference*, 1990.

[19] Hewlett-Packard. *C and C++ SoftBench User's Guide.*

[20] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions of Programming Languages and Systems*, 12(1):26–60, January 1990.

[21] K. Kennedy. A Survey of Data Flow Analysis Techniques. In N. D. Jones and S. S. Muchnick, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, 1981.

[22] K. Kino, S. Yamamoto, and K. Agusa. Abstract Executor for Program Behavior Understanding using the Tool Platform Sapid (in Japanese). In *JSSST FOSE '96*, pages 98–101, Dec 1996.

[23] A. Krishnaswamy. Program Slicing: An Application of Object-oriented Program Dependency Graphs. Technical Report TR94-108, Dept. of Computer Science, Clemson Univ., 1994.

[24] L. D. Larsen and M.J. Harrold. Slicing Object-Oriented Software. In *Proceedings of the 18th International Conference on Software Engineering*, March 1996.

[25] Mary E.S. Loomis. *Object Databases: The Essentials.* Addison-Wesley, 1995.

[26] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[27] K. Ohzaki, S. Yamamoto, and K. Agusa. Dependency Viewer for Program Understanding using the Tool Platform Sapid (in Japanese). In *JSSST FOSE '96*, pages 34–41, Dec 1996.

[28] K.J. Ottenstein and L.M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984. *SIGPLAN Notices* 19(5).

[29] Reasoning Systems. *REFINE User's Guide*, 1994.

[30] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[31] Sapid Home Page. http://www.sapid.org/.

[32] Sun Microsystems, Inc. *The Java Language: An Overview.* http://java.sun.com/docs/overviews/java/java-overview-1.html.

[33] Sun Microsystems, Inc. *The Swing Connection.* http://java.sun.com/products/jfc/tsc/index.html.

[34] Daniel Tkach and Richard Puttick. *Object Technology in Application Development 2nd Edition.* Addison-Wesley, 1996.

[35] Arthur van Hoff, Sami Shaio, and Orca Starbuck. *Hooked on Java(tm).* Addison-Wesley, 1995.

[36] Lois Wakeman and Jonathan Jowett. *PCTE: The Standard for Open Repositories.* Prentice Hall International (UK), 1993.

[37] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[38] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press. Addison-Wesley, 1990.

[39] John Zukowski. *Java AWT Reference.* O'Reilly & Associates, Inc., 1997.

Index