Vol. 40 No. 3 Transactions of Information Processing Society of Japan Mar. 1999

Technical Note

Some Methods for Specializing Object-oriented Programs

YOSHINARI HACHISU,† SHINICHIROU YAMAMOTO†† and KIYOSHI AGUSA†††

In this paper, we propose three approaches to specializing object-oriented programs. The first replaces dynamically dispatched method invocation with statically dispatched method invocation. We describe two techniques for this purpose: the *unique name* method and *rapid type analysis*. The second approach consists of **class fusion** and **class reduction**, which are techniques for merging closely associated classes into a single class. The last approach is **class slimming**, which removes unused methods and variables from classes. These approaches resolve the dilemma of whether to write a program elegantly for easy maintenance or tune it up for good performance.

1. Introduction

Software development based on object-oriented technology has become very popular. But not all object-oriented programs run efficiently in comparison with traditional procedural programs, because of dynamic method invocation, encapsulation, which prohibits the use of specialization methods between classes, and so on. Use of a specializer makes it possible to write a program elegantly and run it efficiently.

In this paper, we propose three approaches to specializing object-oriented programs. The first replaces dynamically dispatched method invocation with statically dispatched method invocation. We describe two techniques for this purpose: the *unique name* method[1] and *rapid type analysis*[2]. The second approach consists of **class fusion** and **class reduction**, which are techniques for merging some classes. The last approach is **class slimming**, which removes unused methods and variables from classes. These approaches resolve the dilemma of whether to write a program elegantly for easy maintenance or to tune it up for good performance. We also show their effectiveness through some experiments.

† Department of Information Engineering, School of Engineering, Nagoya University
†† Faculty of Information Science and Technology, Aichi Prefectural University
††† Center for Information Media Studies, Nagoya University

2. Specializing Object-oriented Programs

2.1 Intra- and Inter-class Specialization

We classify specializing 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 invocation 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, with 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 specialization based on structure between classes. In this section, we propose three approaches to inter-class specialization. The first approach is to replace dynamically dispatched method invocation with statically dispatched method invocation. Many techniques have been proposed for this purpose. In this paper, we describe two techniques: the *unique name* method[1] and *rapid type analysis*[2]. The second approach consists of **class fusion** and **class reduction**, which are techniques for merging closely associated classes. They make it possible to apply intra-class specialization to more classes. The last approach is class slimming, which removes unused methods and variables from classes of class libraries or from classes enlarged by class fusion.

```
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 p = new B();
13:         int r1 = p.foo(2);  // UN, RTA
14:         A q = p;
15:         int r2 = q.foo();   // RTA
16:         System.out.println("r1: "+ r1);
17:         System.out.println("r1: "+ r2);
18:     }
19: }
```

Fig. 1 Static analysis of invocation.

2.2 Static Analysis of Dynamic Invocation

In this section, we describe two techniques for static analysis of dynamic invocation, the *unique name* method[1] and *rapid type analysis*[2].

When a method has a unique name (really a unique signature*) in a program, invocations of it are statically bound. This technique is called the unique name method.

Rapid type analysis searches an entire program for actually instantiated objects. Only methods of classes that have instances are invoked. In this paper, an entire program means a program that includes all required classes and methods. These are used and invoked from the start-up method. We obtain an entire program by class slimming (see Section 2.4).

In **Fig. 1**, `p.foo(1)`, which produces `r1`, can be resolved by the unique name method and rapid type analysis. `q.foo()`, which produces `r2`, can be resolved only by rapid type analysis, because no object of class `A` is instantiated. Rapid type analysis is more powerful than the unique name method.

2.3 Class Fusion

We propose an approach for merging closely associated classes into a single class, which we call **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 produced only to match interfaces. In this case, we can merge the adapter and adaptee classes (**Fig. 2**).

Let us take another example. Assume that, in the design phase, we design class A as an aggregation of classes B and C. In the implementation phase, instances of classes B and C are member variables of class A, and classes B and C are used only in class A. In this case, classes B and C can be merged into class A (**Fig. 3**).

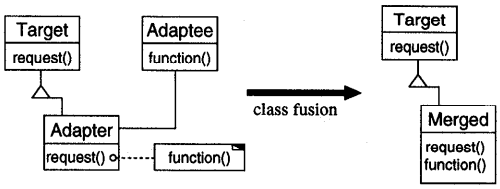

Fig. 2 Class fusion (adapter pattern).

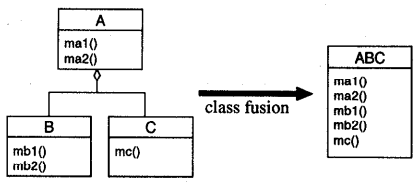

Fig. 3 Class fusion (aggregation of classes).

We formalize class fusion as follows:

Definition 1 (Class Fusion) We can merge class B into class A if

(1) in an entire program, class B is used only in class A, and

(2) class A has a member variable that is an instance of class B.

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

Class B is merged into class A as follows (**Fig. 4**):

(1) A member variable of class A, which is an instance of class B, is removed.

(2) Member variables and methods of class B are added to class A. When their names conflict with members of class A, they are replaced. Method invocations and variable references with instances of class B are replaced with direct access. For example, `adp.function()` is replaced with `function()`.

(3) Constructors of class B are translated into methods and added to class A. For example, in Java, 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()`.

2.3.1 Class Reduction

We extend the idea of class fusion to inheritance classes, merging a class and its subclasses. We call this **class reduction** (**Fig. 5**).

Definition 2 (Class Reduction) We can

* In object-oriented language, a signature means a 3-tuple (method name, parameter types, return type).

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

Fig. 4 Class fusion (sources of adapter pattern).

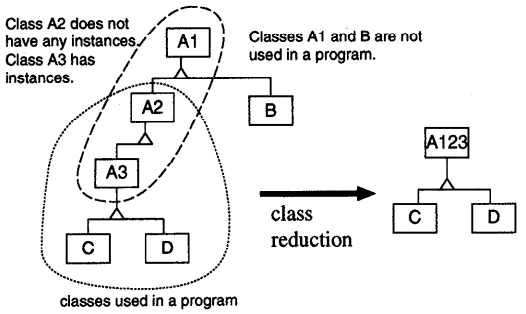

Fig. 5 Class reduction.

merge classes $A_1, A_2, \ldots, A_{n-1}$ into class A_n if (1) there are inheritance relations $A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n$. *Base* $\rightarrow$ *Derived* means that the class *Base* is a superclass of the class *Derived*.
(2) classes $A_1, A_2, \ldots, A_{n-1}$ have no instances in an entire program, and
(3) there is no subclass of class A_i other than A_j $(j > i)$ in an entire program.
We call this merging **class reduction.** □

2.4 Class Slimming

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

Definition 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. □

Classes obtained by class slimming with respect to the start-up method include only methods, and variables necessary for a program. This is an entire program that is used for static analysis of dynamic invocation, and for class fusion and reduction. Class slimming is also useful for removing unused methods from classes enlarged by class fusion and reduction.

Class slimming is performed by using a call graph from method m. However, creating a call graph statically is difficult, because of dynamically dispatched method invocation. We have therefore modeled the process by using control, data, and object flows.

Table 1 Applicability of class fusion and reduction.

Program	Number of Classes			
	Total	Inh.	Fusion	Red.
JDK Demos	139	88	9	9
Bubble Sorting	40	24	7	7
Grep	52	24	4	6

Inh.: Inheritance, Red.: Reduction

3. Evaluation

In this section, we describe two experiments that we carried out to demonstrate the effectiveness of our approach: the first shows the applicability of class fusion and reduction, and the second shows the speedup obtainable by class fusion and reduction. We created some tools for the experiments by using a CASE tool platform, Japid[3].

The first experiment showed the extent to which class fusion and reduction can be applied to practical programs. We selected JK Ddemos, grep with a class library of regular expressions, and integer sorting by means of the bubble sorting algorithm☆ (**Table 1**). JDK demos include six programs and use 139 classes, which include classes of JDK's Java class library. The classes have 88 inheritance associations. We found nine classes for fusion and nine classes for reduction. Grep and bubble sorting were carried out in the same way.

☆ 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. To clarify the effect of specialized parts and to hide the overhead of reading a file, we selected the bubble sorting algorithm.

Table 2 Speedup (bubble sort).

Case	Time [sec.]	Speedup
Not merged	52.5	1.00
1 reduction	42.4	1.23
1 fusion and 1 reduction	42.4	1.23

Table 3 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

The second experiment showed the speedup that can be obtained through the use of class fusion and reduction. Classes were merged manually and compiled with an optimization option (`-O`). We selected bubble sorting (**Table 2**) and grep (**Table 3**).

In the case of bubble sorting, an unspecialized program sorts five thousand integers in 52.5 seconds. Programs to which class fusion and reduction were applied completed sorting in 42.4 seconds and 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 their byte code (**Tables 4** and **5**). In the case of bubble sorting, the number of method invocations, particularly dynamic method invocations, were reduced. In our program, to change an ordering rule, a method for comparing two numbers is defined as an abstract method, which must be overridden by a subclass. In a program to which class reduction was applied, it was recognized as a static invocation and was in-lined, resulting in a 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.

4. Conclusions

In this paper, we have described three approaches to specializing object-oriented programs: static analysis of dynamic invocation, class fusion and reduction, and class slimming. We demonstrated their effectiveness by showing that class fusion and reduction are applicable to practical programs, and that a specialized program runs 23 percent faster than an unspecialized one. Owing to class fusion and reduction, a dynamic method invocation is replaced with a static one and is in-lined. In another case, however, class fusion and reduction do not result in any speedup. We must therefore find a method for evaluating the effect of class fusion and reduction before they are applied.

In future, we plan to examine the applicability and effects of class fusion and reduction for larger programs. We also plan to model and implement class slimming, and to implement a specializer by using Japid.

Table 4 Analysis of byte code (bubble sorting).

Case	Byte Code [lines]	Method Invocations Dynamic / Total
Not merged	244	25/44
1 reduction	253	22/39
1 fusion and 1 reduction	246	22/36

Table 5 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

References

1) Calder, B. and Grunwald, D.: Reducing Indirect Function Call Overhead in C++ Programs, *POPL '94*, pp.397–408 (1994).
2) Bacon, D.F. and Sweeney, P.F.: Fast Static Analysis of C++ Virtual Function Calls, *OOPSLA '96*, pp.324–341 (1996).
3) Hachisu, Y., Yamamoto, S., Hamaguchi, T. and Agusa, K.: A Fine Grain Source Repository for Java (in Japanese), *Proc. Computer System Symposium*, pp.147–154 (1996).

(Received August 31, 1998)
(Accepted December 7, 1998)