Object-Oriented Specialization Using Aspect Programming

Adrián SLAVKOVSKÝ*
Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 3, 842 16 Bratislava, Slovakia
adrian.slavkovsky@gmail.com

Abstract. Object-oriented programming using design patterns provides reusability and genericness of software products but causes overheads what reflects in efficiency. These overheads can be partially eliminated by specialization. One way how to link original and specialized code together is to use aspect-oriented programming. This paper analyzes various means of forming specialized aspect codes into original code and describes advantages and disadvantages joined with particular approaches as well as results of specialization improvement being achieved with them.

1 Introduction

Design patterns in object-oriented approach present a major step towards defining, formalizing and solving common design problems in software architecture. But their generality brings inefficiency into the programs. Objects are accessed mainly through interfaces or abstract classes so there are many virtual calls which have to be evaluated runtime. Standard compiler optimizations do not fully eliminate these kinds of overheads. Therefore a program specialization was proposed to solve this problem.

There are many ways how to use specialization. This paper focuses on object-oriented specialization especially on eliminating virtual calls. Schultz et al. have defined what to specialize, how we should specialize and where specialization is effective [5, 6]. He defined a structure of specialization patterns as a complement to design patterns [7]. Schultz has also described different ways of integration of original and specialized code. One option is to use aspect-oriented programming. Our work shows some possibilities of how can aspect code be used during the specialization

* Supervisor: prof. Ing. Pavol Návrat, PhD., Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

Two different concepts of aspect interaction were proposed – static, which is made by introduction and is woven into original code in compile time and dynamic, which allows us to replace virtual calls with our specialized code when particular call occurs. At the end specialization example and results are shown which document my observations of specialization improvement on decorator design pattern.

2 Program Specialization

Specialization is a technique of code transformation. Primary goal of this transformation is to produce more effective code than the original one. To achieve this, we need to know some specific information about the context this code is running in. In object-oriented programming, virtual dispatching represents one source of overhead. Virtual calls make program control flow more difficult and block traditional compiler and hardware optimizations, because compiler often does not know the concrete type of object.

Specialization is supposed to be a complement to compiler optimization but it is driven explicitly by programmer. He is the one, who chooses the appropriate fragment of code to specialize and he defines invariant information about fragment's context (which compiler usually doesn't know) that means he in fact directs the specialization.

3 Specialization patterns

Using design patterns simplifies process of software design and development, but bears respective overhead. Schultz et al. proposed specialization patterns as a complement to design patterns in terms of defining the specialization opportunities [7]. Each design pattern can have corresponding specialization pattern. This pattern contains information where and what kind of overhead is presented, which parts can be specialized to eliminate this overhead, what are the conditions of execution of specialization, specialization strategy, proposal of specialization class and applicability. After determining the right type of specialization we specialize selected code fragments. The last part of specialization process is integration of original and specialized code. One way how to make it is to put specialized methods into original classes. The problem of this concept is to ensure security. Another approach is to encapsulate specialized code into separated classes. Specialized methods from separate class will not be able to access private or protected attributes of original class. Integration can be also achieved using aspect-oriented programming. Specialization code is separated in aspect and woven into original code in compile time. Thus we can distinguish between original and specialized code and if we decide not to use specialization we simply won't weave aspect code into original.
4 Specialization using aspect programming

Aspect-oriented programming is an approach for realizing separation of concerns by grouping them into logical units. In compilation process aspect code is first woven into original by aspect weaver. Resulting code is then compiled by regular compiler.

4.1 Static approach

This concept simply adds new methods and attributes into existing code in compile time (more precisely in weaving time). The place of addition is defined by pointcut expression. Aspect programming uses term introduction. After specialized methods are introduced into original program the client code invoking original code needs to be changed. Change consists of replacing these calls with specialized ones.

4.2 Dynamic approach

This concept is rather different from the static one. It is based on dynamic activation of advice code whenever specific code join point is reached. We can bind this advice code to a code pointcut what in a case of specialization may represent virtual call. If we want to replace call of original function we use an around section with specialized code and omit a statement which calls it. Advantage of this approach is that after specialization we don't need to change any client code. Aspect containing specialized code is activated when relevant code join points are reached. Unfortunately this approach has limited applicability in present time because AspectC++ produces some overhead when invoking advice code. This overhead grows with number of parameters that need to be passed to specialized methods and as my tests show it costs more than the call to the virtual function. When this problem will be solved we can expect that this type of specialization will be fully applicable.

5 Specialization example

Example consists of specialization of program using design pattern Decorator [1]. The overhead in this case is coupled with method invocation on AComponent object. Client is not aware of concrete type of object being hidden behind this abstract class. We are able to eliminate virtual methods in fragments in which we can determine the concrete type of the object the client is working with. For this purpose let us say that it is CConcreteDecoratorA, which has CConcreteComponent as inner component.
AComponent *comp = new CConcreteComponent();
//before specialization
AComponent *dec = getDecorator(comp); //get Decorator
result = dec->doSomething(a);
aspect Decor { //static specialization
  pointcut spec() = "CConcreteDecoratorA";
  advice spec() : long int doSpecSomething(int a) { 
    before();
    CConcreteComponent* comp = (CConcreteComponent *)
      this->getComp();
    num = comp->doSomething(a);
    after(); return num; }
  result = cDecA->doSpecSomething(); //specialized client
}

aspect Decor { //dynamic specialization
  advice args(a) && call("% AComponent::doSomething(...)"
    : around(int a) {
    CConcreteDecoratorA *cDec = (CConcreteDecoratorA *)
      tjp->target();
    CConcreteComponent* comp = (CConcreteComponent *)
      cDec->getComp();
    cDec->before();
    num = comp->doSomething(a);
    cDec->after(); *tjp->result() = num; }
}

In static approach we will introduce new specialized method into
CConcreteDecoratorA to ensure that it has access to all attributes and methods of
this class. In dynamic specialization we need to obtain a parameter of original function.
Then we get a reference to target of the call (we know the type of object) and reference
to its inner component.
6 Results

Tests were made using ac++ 0.9 under Microsoft Visual C++ .NET in cooperation with AspectC++ Add-In under two configurations – Debug mode and Release Mode (full optimizing mode). Specialization using introduction accelerates program from 2 to 15%. Using dynamic concept we have observed slowdown of specialized program. Because decorator contains the other component (which could be decorator as well) specializing a method of such object can be done via recursive algorithm.

```
unfoldDecorator(AComponent *dec) {
    if (dec is not component) {
        dec->before(); //do before method
        unfoldingDecorator(dec->getComponent());
        dec->after(); //do after method
    } else return dec->doMethod();
}
```

To improve performance and eliminate excessive method invocation we can provide references to component objects as parameters of specialized function and call methods on these objects in function body respectively. The results are shown in tab. 1.

<table>
<thead>
<tr>
<th>Compile mode</th>
<th>Specialization description</th>
<th>Function call</th>
<th>% speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>method with parameter</td>
<td>doSpecSomething(a)</td>
<td>1.2</td>
</tr>
<tr>
<td>Debug</td>
<td>method with parameter and pointer to component of decorator</td>
<td>doSpecSomething(a, pointer)</td>
<td>4.9</td>
</tr>
<tr>
<td>Debug</td>
<td>method without parameter</td>
<td>doSpecSomething()</td>
<td>5.1</td>
</tr>
<tr>
<td>Debug</td>
<td>method without parameter and with pointer to component</td>
<td>doSpecSomething(pointer)</td>
<td>11</td>
</tr>
<tr>
<td>Debug</td>
<td>recursive decorator</td>
<td>doSpecSomething()</td>
<td>15.3</td>
</tr>
<tr>
<td>Release</td>
<td>method with parameter</td>
<td>doSpecSomething(a)</td>
<td>0.9</td>
</tr>
<tr>
<td>Release</td>
<td>method with parameter and pointer to component of decorator</td>
<td>doSpecSomething(a, pointer)</td>
<td>1.4</td>
</tr>
<tr>
<td>Release</td>
<td>method without parameter</td>
<td>doSpecSomething()</td>
<td>1.6</td>
</tr>
<tr>
<td>Release</td>
<td>method without parameter and with pointer to component</td>
<td>doSpecSomething(pointer)</td>
<td>3</td>
</tr>
<tr>
<td>Release</td>
<td>recursive decorator</td>
<td>doSpecSomething()</td>
<td>5.9</td>
</tr>
<tr>
<td>Debug</td>
<td>dynamic specialization</td>
<td></td>
<td>-3.6</td>
</tr>
<tr>
<td>Release</td>
<td>dynamic specialization</td>
<td></td>
<td>-1.5</td>
</tr>
</tbody>
</table>

From these observations we can say that in present time the only applicable way how to integrate specialized and original code with aspect programming is to use...
introduction which gives us promising results and speedups. In future there is a possibility to use dynamic approach when related overhead produced by aspect will be eliminated.

7 Conclusion

Object-oriented programming is dominating paradigm in developing commercial applications. With aim of design patterns software becomes more generic, reusable, modular and adaptable but leads also to performance expenses. These overheads can be eliminated by specialization. This paper focuses on object-oriented specialization and integration of original and specialized code using static and dynamic aspect approach. The results of my observations show that static way accelerates program's execution. The limitation of dynamic way lies in overhead internally caused by aspect language. If these overheads shall be eliminated this approach could be more perspective.

Acknowledgement: This work has been partially supported by the Grant Agency of Slovak Republic grant No.VG1/0162/03.

References