Optimizing Object-Oriented Languages Through Architectural Transformations

Tom Tourwé* and Wolfgang De Meuter

{Tom.Tourwe,wdmeuter}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

Abstract. Certain features of the object-oriented paradigm are a serious impediment for the runtime performance of object-oriented programs. Although compiler techniques to alleviate this problem were developed over the years, we will present some real-world examples which show that these solutions fall short in making any significant optimizations to systems that are required to be very flexible and highly reusable. As a solution, we propose a radically different approach: using an open compiler to "compile away" whole designs by performing architectural transformations based on programmer annotations. We will discuss this approach in detail and show why it is more suited to solve the efficiency problems inherently associated with object-oriented programming.

1 Introduction

It is well known that certain distinguishing features of the object-oriented paradigm are a serious impediment for the runtime performance of an object-oriented system. The most important and powerful feature of object-oriented programming languages which is hard to implement efficiently is polymorphism, or the ability to substitute any object for another object which understands the same set of messages in a certain context. Due to this feature, a compiler cannot predict at compile time which method will be executed at runtime by a particular message. Thus, so called *dynamic dispatch* code has to be generated for message sends, which looks up the appropriate method based on the runtime type of the receiver. In comparison to code for a normal function call, dynamic dispatch code is clearly much slower. Apart from this *direct cost*, polymorphism is also responsible for the fact that traditional compiler optimizations can no longer be performed. For example, inline substitution of method bodies is no longer possible, because the compiler cannot statically determine which method will be invoked by a particular message. This is called the *indirect cost*. It is stated in [6] that some programs spend up to 47 % of their time executing dynamic dispatch code, and that this number is still expected to increase in the

* Author financed with a doctoral grant from the Institute for Science and Technology (IWT), Flanders.

future. One of the main reasons thereof is that object-oriented programming encourages a programmer to write many small methods (a few lines of code) that get polymorphically invoked by each other. Unfortunately, recent insights in the programming style fostered by the object-oriented paradigm precisely encourage the use of these features. Proof thereof is the tremendous success of programming conventions such as design patterns [7] and idioms [1, 3].

Of course, the reason for the success of these programming techniques is that there currently exists a trend to make a software system comply to many important non-functional requirements, such as reusability, extendability and adaptability, enabling the developers to reuse major parts of it. This leads to systems in which a lot of attention is paid to the global *architecture*. How the different classes in a system are combined and the specific ways in which their objects interact becomes very important in order to be able to easily reuse or extend the system. The techniques that support fulfilling this goal however, encourage the use of the specific features of object-oriented languages even more. Thus, more often than not, system developers are confronted with a dilemma: should a system be written in a very flexible and highly reusable way (thereby heavily relying on late binding polymorphism), which may lead to inefficient code, or should they take into account the efficiency of a system and not care about the non-functional aspects of the code? As a result, developers are often tempted to avoid using inefficient features, which clearly does not contribute much to the quality of the software.

Not surprisingly, techniques have been developed over the years which focus on trying to eliminate dynamic dispatch code. This is achieved by trying to predict the exact type an object will have at runtime, which then allows the compiler to statically bind the messages sent to this object. Although the results are encouraging, we will argue why these techniques in isolation are not sufficient to significantly optimize future object-oriented systems. The main deficiency is that they only have a narrow local view of the system. We will show that they fail to incorporate global knowledge about the system's structure and architecture and are therefore forced to make local and more conservative optimizations. To alleviate this problem, we propose to use an open compiler, which is able to reason about programs at the meta level and which can perform architectural transformations based on the information gathered this way. The compiler is open so that developers can annotate their source code and provide the compiler with detailed architectural knowledge about their system. Further, it is able to reason about a system at the meta level in order to deduce even more knowledge about its architecture.

The paper is structured as follows. The next section discusses existing techniques for improving the performance of object-oriented systems, while section 3 provides a representative example and an in-depth discussion as to why these techniques on their own are not capable to improve performance of (future) object-oriented systems significantly. In section 4, we present our approach and explain the framework we use for reasoning about programs and using programmer annotations. Section 5 explains how our approach enables significant optimization of, amongst others, the example introduced in section 3, while section 6 describes future work and section 7 concludes.

2 Current Optimization Techniques

This section presents some of the most important techniques developed to overcome the efficiency problem of object-oriented languages. More specifically, *class hierarchy analysis, customization* and *exhaustive class testing* will be discussed. We present the overall picture and elaborate only on the properties needed to understand the discussions in the following sections. We refer the reader to [5, 2] for detailed descriptions of these techniques and a detailed report of the results.

2.1 Class Hierarchy Analysis

Class hierarchy analysis tries to avoid dynamic dispatch code by providing the compiler with the class hierarchy of the whole program. It is based on the observation that sometimes, when compiling a class, knowledge about its superclasses and subclasses can help in determining which method will be invoked by a particular message send. An example will make this more clear. Consider the inheritance hierarchy in Figure 1 and suppose that method `p` of class `G` performs a self send of the message `m`. When straightforwardly compiling the method `p`, dynamic dispatch code will have to be generated for this message send, since there are different implementations of the method `m` in the hierarchy. However, if the compiler takes into account the class hierarchy of class `G`, it is able to statically bind the message. The method `m` is never overridden: not in class `G`, nor in any of its subclasses. Thus, the method `m` that will be executed at runtime by method `p` is the one that is defined in class `C`.

2.2 Customization

Customization statically binds message sends by compiling different versions of the method in which they occur. Each version is specialized for one particular receiver. Since the receiver is thus statically bound in each of these different versions, the compiler is able to avoid dynamic-dispatch code for each self send occurring in that method.

Consider again the class hierarchy in Figure 1. The method `o` of class `B` has three different types of possible receivers: the classes `B, D` and `E`. Thus, the method `o` is compiled to three different versions, corresponding to each of the three possible receivers. In each of these three versions, the receiver of the message is statically bound. If method `o` of class `B` performs a self send of message `m`, the specialized method `o` for class `B` will bind this message to method `m` of class `B` and the specialized version for class `D` will bind this message to method `m` of class `D`. Note that class hierarchy analysis would not be able to statically bind this self send, since the method `m` is overridden in subclass `D` of `B`.

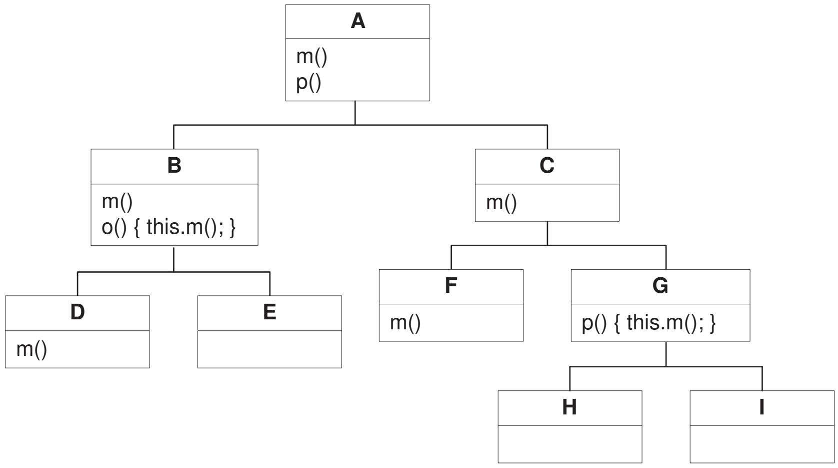

Fig. 1. Class `A` and its subclasses.

A disadvantage of this technique is the risk of code explosion: when a large number of classes are possible receivers for a certain message, many specialized versions of the corresponding method need to be generated. Also, methods are only specialized for the receiver of a message, which only enables statically binding self sends. Other message sends occurring in the method body, for example messages sent to an argument or to an instance variable, are not taken into account. For these reasons, an extension of the customization technique, called *specialization* was developed. However, the problem of code explosion becomes even worse using this technique, since even more different versions need to be generated for a method. Also, specialization requires support for multi-methods in the runtime environment, something popular languages as C++ or Java lack.

2.3 Exhaustive Class Testing

Exhaustive class testing is a technique which statically binds a message by inserting appropriate class tests around it, taking into account the class hierarchy. Consider again the hierarchy in Figure 1. Instead of specializing the method `o` for each of its possible receivers, the compiler could generate the following code for the message `m`:

```
if(receiver instanceof D) { <code of method m in class D> }
else if(receiver instanceof B || receiver instanceof E} {
   <code of method m in class B>
}
else { receiver.m(); } // send the message
```

At first sight, it might seem that this code can be optimized even further by using a switch-statement (e.g. a dispatch table). This is however not the case, since most of the time this table would be sparse. The problem of finding a numbering scheme for the classes in the system so that for every message send a dense table can be constructed is very hard, or even impossible. For this reason, class testing is very efficient only if the number of possible receivers for a message send is small and if the execution of dynamic dispatch code is more costly then the execution of a class test and all tests that failed before. This means that there can be a loss in performance when there are many possible receivers, since many tests will have to be performed. Furthermore, it is always possible that the same method is executed each and every time, which means that the other tests will never succeed, but they still have to be executed.

2.4 Conclusion

This section presented some of the techniques developed recently in order to alleviate the efficiency problem of object-oriented languages. The primary focus of all these techniques lies in trying to avoid the generation of expensive dynamic dispatch code. This is achieved by statically binding as much message sends as possible. An important observation that can be made is that no one technique is able to solve the problem on its own. Depending on the situation and the context of a message send, one certain technique is better suited than another.

3 An Illustrating Example

In this section, we will present a representative example which shows how writing software in a flexible and reusable way often incurs an inherent performance loss. Also, it will become clear that, in order for software to be reusable, heavy reliance on late binding polymorphism is required. Furthermore, we will point out why the techniques discussed in the previous section are not sufficient to significantly optimize code for this and other examples.

A generally accepted collection of techniques for writing reusable and adaptable software today is design patterns [7]. Given their popularity and widespread use, it is extremely relevant to discuss the efficiency of systems using these patterns. The example presented here is thus taken from [7]. For a more in-depth discussion of the use, the advantages and the disadvantages of design patterns, we refer the reader to this book.

3.1 The Visitor Design Pattern

Problem Statement and Solutions. The prototypical example of the use of the Visitor pattern is the architecture of a compiler that needs to traverse the abstract syntax tree (AST) of a program many times in order to pretty-print, typecheck or generate code for it. Instead of implementing these operations on the elements that make up the AST, as is depicted in Figure 2, we could implement each operation in a different class, called a *Visitor* class, and pass objects of this class to the elements of the AST. These elements then call the appropriate method in the Visitor object. This solution is depicted in Figure 3. This way of defining operations on element classes has several advantages. Clearly, the code becomes much easier to understand, maintain and reuse, since element classes are not cluttered with code for different operations. Also, adding new operations becomes much easier, as this simply boils down to implementing the appropriate subclass of the Visitor class and no element classes need to be changed.

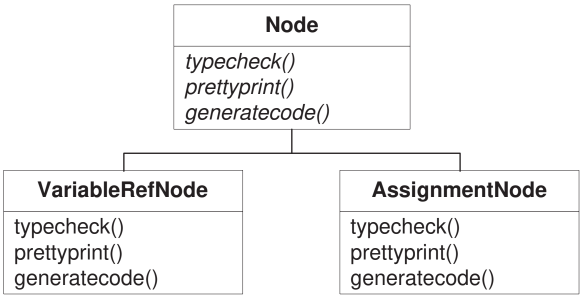

Fig. 2. The straightforward solution.

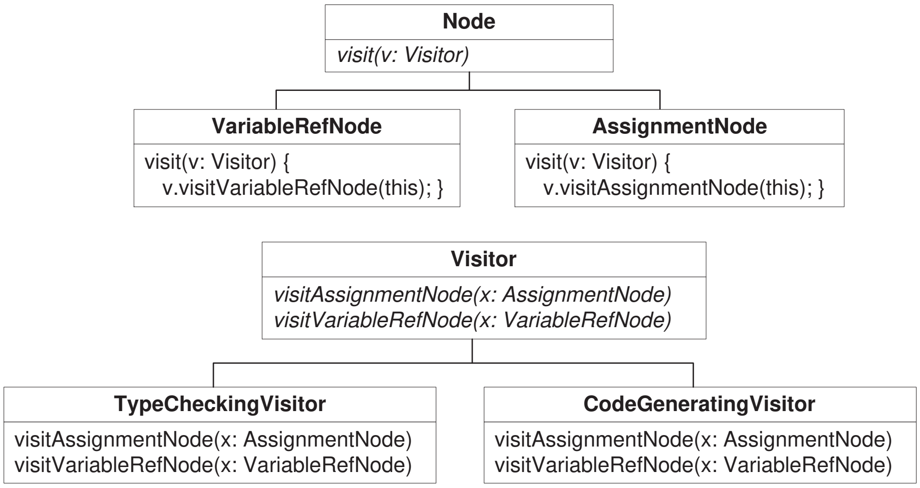

Fig. 3. The Visitor design pattern solution.

The Visitor pattern relies on a technique called *double dispatch* [1]. By using this technique, one can ensure that the method that will eventually get executed by a message send not only depends on the runtime value of the receiver, but also on the value of the message's argument. The `visit` method in the `Node` class and its subclasses is an example of this technique. The method that will get called is dependent on both the type of the receiver of the `visit`-method and the type of the `Visitor` argument passed to it.

Why Current Optimization Techniques Fail. Clearly, since current optimization techniques only focus on statically binding the receiver of the message, they fall short in optimizing methods that use double dispatch. In order to arrive at the method that will perform the actual work, at least two message sends need to be statically bound. This is not a trivial task, since these two messages are nearly always subject to late binding, because many subclasses of the `Node` and the `Visitor` classes exist. Often, simply sending the message will thus be more efficient then inserting many class tests or compiling many specialized methods. A more important reason for the failure of these techniques, however, is that they do not take into account the specific architecture of this pattern. The whole idea behind it is to separate the operations that classes need to define from these classes themselves. As a result, many more messages need to be sent. Instead of trying to statically bind these messages, a better approach would consist of avoiding these message sends altogether. A significant performance gain can be achieved, for example, by moving the operations back to the specific subclasses of class `Node`, avoiding at least half the total number of messages sent[1]. The result of this operation would then be the architecture depicted in Figure 2. Of course, as already mentioned, this architecture is much less flexible and reusable. Therefore, the compiler should be able to perform the transformation of the architecture automatically, so that developers can still produce flexible code. Current techniques do not make this possible, however.

3.2 Conclusion

Similar observations can be made for other design patterns, but other examples were left out due to space limitations. These observations lead us to conclude that compliance to many important non-functional requirements, such as reusability and flexibility, incur an inherent performance loss. The primary reason for this is that developers have to rely heavily on features which are hard to compile efficiently. Furthermore, current compiler techniques are not capable of optimizing such systems significantly, as the example we presented clearly showed. This is because those techniques focus only on statically binding messages and do not incorporate a more architectural view of a system. Design patterns and other techniques for writing flexible software often introduce extra abstractions and make clever use of inheritance and delegation in order to make objects more interchangeable and to prepare for future extensions. Since current techniques

[1] In reality, the performance gain will even be higher as it is a characterizing property of a visitor to access the state variables of the visited object through accessors. These can also be removed easily.

fail to recognize these extra abstractions, it is clear that they are not able to eliminate them, thereby reducing the number of message sends. For a compiler to be able to optimize such systems, it should incorporate some techniques to transform one architecture into another more efficient one, eliminating redundant message sends instead of trying to statically bind them.

4 Architectural Optimization Through Transformations

The main reason why current compilers are not able to significantly optimize highly flexible systems is because they cannot automatically infer the intentions of the developer. A compiler does not know, for example, why a specific abstraction is introduced, so it cannot eliminate it to produce better code. Therefore, in our approach, these intentions are made explicit. For this purpose, an *annotation* language is provided in which these intentions can be expressed. In order for the compiler to be able to use this information in a useful way, it should incorporate some knowledge on how to optimize a certain intention. Again, this knowledge should be provided by the developer and can be expressed in the *transformation* language.

A drawback of our approach is that a lot of user intervention is required for the optimization of a system. We believe this is unavoidable however, as systems tend to get more complex and because there are limits to the amount of information that can be deduced automatically by dataflow analysis techniques [11]. It should be stressed however, that we strive to minimize developer intervention as much as possible. First of all, some work has been published recently in the area of the automatic detection of design principles, such as design patterns [10]. By integrating this work into our framework, the burden of manually specifying intentions becomes obsolete. Second, popular techniques for constructing flexible systems, such as design patterns, are used over and over again. This means a library of commonly used transformations can be constructed, which can be reused for compiling different systems and which releaves developers from specifying the same transformations over and over again. Furthermore, the fact that the intentions of a developer are made explicit in the software can also aid in other areas besides performance. First of all, documentation and understandability can be improved upon. Second, the information revealed by the intentions can be used to study the evolution conflicts of object-oriented systems in more detail [9].

It is important to note that the transformations performed by our compiler are source-to-source transformations. This has two main advantages. First of all, the code our compiler outputs can still be optimized by current optimization techniques in order to achieve even better performance. Second, current techniques will benefit from the fact that unnecessary abstractions are removed by our compiler, as this allows them to statically bind even more messages.

In what follows, we will explain how all these different aspects can be integrated into one uniform framework which allows for architectural optimization of object-oriented systems.

4.1 A Uniform Framework

The transformational approach we propose poses some important requirements. In order to be able to specify and perform the transformations on a program easily, a suitable representation of it has to exist. Surely, the transformations can be performed on the abstract syntax tree of the program, although it has been argued that this representation is not very well suited for this purpose [8]. Also, the representation should be designed in such a way that it allows for easy recognition of specific (user defined) patterns in a program, which the AST certainly is not. The work done in [10] and the conceptual framework introduced in [4] inspired us to represent a program in a special-purpose logic language. For our purposes, the declarative nature of such languages and the builtin matching and unification algorithms are very important features.

Following [4], we represent a program in the logic language *TyRuBa* by means of a set of logic propositions. How constructs of the base-language (the language in which the program is implemented) are represented in the logic language is specified by the *representational mapping*. We will discuss this in more detail below. A *code generator* is associated with the representational mapping, specifying how code should be generated from the logic propositions representing the program. It can be seen as the inverse function of the representational mapping.

We will now describe some important properties of TyRuBa, the logic language we use, and will then continue explaining in more detail how a base-level program is represented in this language and what the annotation and the transformation language look like.

TyRuBa. The TyRuBa system is basically a simplified Prolog variant with a few special features to facilitate code generation. We assume familiarity with Prolog and only briefly discuss the most important differences.

TyRuBa's lexical conventions differ from Prolog's. Variables are identified by a leading "`?`" instead of starting with a capital. This avoids confusion between base-language identifiers and variables. Some examples of TyRuBa variables are: `?x`, `?Abc12`, etc. Some examples of constants are: `x`, `1`, `Abc123`, etc. Because TyRuBa offers a quoting mechanism which allows intermixing base-language code and logic terms, the syntax of terms is also slightly different from Prolog's. To avoid confusion with method calls, TyRuBa compound terms are written with "`<`" and "`>`" instead of "`(`" and "`)`".

TyRuBa provides a special kind of compound term that represents a piece of "quoted" base-language code. Basically this is simply a special kind of string delimited by "`{`" and "`}`". Instead of characters however, the elements of such quoted code blocks may be arbitrary tokens, intermixed with logic variables or compound terms. The following is an example of a quoted term, in Java-like syntax. Note the variables and logic terms that occur inside the code.

```
{ void foo() {
    Array<?El> contents = new ?El[5];
    ?El anElement=contents.elementAt(1); }
}
```

The Representational Mapping. The representational mapping specifies how a program is represented in the logic language. This basically means that constructs of the base-language are represented in the logic language by means of a set of logic propositions. The mapping scheme between logic representation and base-language may vary and determines the kind of information that is accessible for manipulation. For our purpose, a fine-grained mapping is necessary, since we want to be able to manipulate a program at every level of detail. Furthermore, more structural and higher-level information, such as the relationship between different classes and their specific interaction, also needs to be modeled in the logic language to enable easy reasoning about the architecture of the program. It should be stressed that the representation of a program is generated automatically by the compiler and that it is thus not the task of the developer.

We will explain this representational mapping by using the following running example:

```
class Test extends SuperTest {
     boolean b;
     int m(int i) {
          if(b)
              return i;
          else
              return 0;
     }
}
```

The presence of a class and its position in the inheritance hierarchy is made explicit by the following facts:

```
class(Test).
extends(Test,SuperTest).
```

Classes are regarded as being composed out of instance variables, methods and constructors. The instance variables of a class are represented as follows:

```
field(Test,boolean,b).
```

The *field* predicate thus always indicates to which class the field belongs and specifies the type and the name of the variable. The presence of methods and constructors is asserted in a different way: their declarations are chopped up into little pieces, each of which represents one particular aspect. It is the responsibility of the code-generator to assemble the various parts and generate code accordingly. The method `m` can be represented as follows:

```
method(Test,m,[int],method1).
returntype(method1,int).
formalparameter(method1,int,i).
body(method1,blocknode).
```

These predicates specify the returntype, the formal parameters and the body of the method. The first argument of each predicate specifies to which method the particular feature belongs. This is necessary for the code generator so that it can assemble all parts of a specific method in order to generate code for it. Note that the *method* predicate not only lists the name of the class and the name of the method, but also the type of the formal parameters. This is needed in order to uniquely identify the method, as it can possibly be overloaded.

Bodies of methods and constructors consist of statements and expressions and can thus be represented by a normal parse tree. The nodes of this parse tree need to be unique, and we need to be able to refer to them in an easy way. Therefore, a proposition representing a node has an extra argument, which specifies its (unique) name. The body of the method `m`, for example, will be represented as follows:

```
blockstatementnode(blocknode,[ifnode1]).
ifstatementnode(ifnode1,expr-node,then-node,else-node).
fieldaccessnode(expr-node,this,b).
returnstatementnode(then-node,ret-expr).
returnstatementnode(else-node,litnode0).
variableaccessnode(ret-expr,i);
literalnode(litnode0,0).
```

As should be clear, all nodes point to their child nodes via their specific names. The `ifstatementnode`, for example, mentions its three child nodes, a condition node, a then node and an else node, by their respective names. Furthermore, the two `returnstatementnode`s picture why nodes need to have a name: two nodes of the same kind can exist and we need to be able to make a distinction between them, because they can occur in different parts of the program. Again, this representation is generated automatically by our compiler.

The Annotation Language. The declarative nature of a logic language allows developers to straightforwardly provide the compiler with architectural knowledge by means of logic facts. When annotating a particular design pattern in a system, for example, the developer should state which classes are the primary participants in the architecture of that pattern. Consider for example the annotated occurrence of a visitor design pattern below:

```
/** ConcreteVisitor(PrettyPrintVisitor).
    operationname(PrettyPrintVisitor,prettyprint).
 */
public class PrettyPrintVisitor extends Visitor { ... }

/** ConcreteElement(AssignmentNode). */
public class AssignmentNode extends Node { ... }

/** ConcreteElement(VariableRefNode). */
public class VariableRefNode extends Node { ... }
```

The assertions, which are embedded in `javadoc`-like comments, state that the classes `AssignmentNode` and `VariableRefNode` are instances of the concrete element participants and that the class `PrettyPrintVisitor` is an instance of the concrete visitor participant of this pattern. The predicate `operationname` specifies the name of the operation that is implemented by the particular concrete visitor. The reason for its presence will become clear soon. Using this information, together with the rules expressing which transformations should be performed, our compiler is able to transform the visitor architecture into a more efficient architecture (this will be shown in section 5). Of course, apart from classes, individual methods, fields and constructors can also be annotated in this way.

An advantage of specifying this information through `javadoc`-like comments is that the source code of the system is not mixed with the annotations that handle its performance. This is important because mixing the code with directives makes it less readable. Also, current compilers will treat annotations as comments and can thus still be used to compile this code, although the result will not be as efficient as when the program is compiled with our optimizing compiler.

The Transformation Language. A transformation that should be performed on a program basically expresses "if this particular pattern occurs then replace it by this pattern". An intuitive way to express a transformation is thus via a logic rule: the condition of the rule corresponds to the condition of the if and the head of the rule corresponds to the then part of the if. Consider the following example:

```
messagenode(?nodename,?rec,?var,[]):-fieldaccessnode(?nodename,?rec,?var).
```

The result of this rule is that the compiler will replace all direct variable references, such as `this.b`, by message sends, such as `this.b()`[2]. Note that the name of the `messagenode` predicate is copied from the `fieldaccessnode` predicate. This is to ensure that the code generator will generate code for the message send only. Remember that node names need to be unique, so the code generator has special provisions so that code is generated only once for a specific node.

5 The Illustrating Example Revisited

In this section, we will show how the example presented in section 3 can be optimized by using architectural transformations. The main idea upon which this revisited example relies is that the complete architecture of the particular design pattern is compiled away. This is achieved by transforming the solution it proposes into a more straightforward solution for the same problem. As already shown in previous sections, the result of this operation will be more efficient, as a straightforward solution is often much less flexible and does not rely on polymorphism as much. As a consequence, much less messages will need to be sent and thus the performance of the system will be improved significantly.

[2] Note that this is only an illustrative example. Expressions of the form `this.b = 2` will be replaced by `this.b() = 2` which is of course not correct.

The Visitor pattern defines an architecture in which Visitor classes implement an operation over some object structure. As already explained, this architecture can be optimized by implementing the operation defined by a specific Visitor class on the elements that make up this object structure. A concrete example should make this more clear. Consider a `visitIfStatementNode` method in a `PrettyPrintVisitor` class, which could look like this:

```
void visitIfStatementNode(IfStatementNode x) {
   this.printOnOutputStream(``if(``);
   x.getCondition().visit(this);
   this.printOnOutputStream(``) '');
   x.getThenPart().visit(this);
   this.printOnOutputStream(`` else ``);
   x.getElsePart().visit(this);
}
```

This method should be moved to the `IfStatementNode` class and the code should be changed so that it looks like this:

```
void prettyprint(PrettyPrintVisitor x) {
   x.printOnOutputStream(``if(``);
   this.getCondition().prettyprint(x);
   x.printOnOutputStream(``) '');
   this.getThenPart().prettyprint(x);
   x.printOnOutputStream(`` else ``);
   this.getElsePart().prettyprint(x);
}
```

A number of changes needs to be made to the first code fragment in order to arrive at the second. First of all, the `visitIfStatementNode` method should be moved to the class of its formal parameter (e.g. the `IfStatementNode` class) and its name should be changed to `prettyprint`. Note that, since the method is moved and not copied, the original `visitIfStatementNode` method will be deleted from the `PrettyPrintVisitor` class. Second, the type of the formal parameter of the method should be changed to the concrete visitor class (e.g. `PrettyPrintVisitor`). This is necessary so that methods and instance variables of this visitor class can still be accessed. Third, since the method is moved to the class of its formal parameter, all references to the formal parameter should be replaced by self-references. All self-references in turn have to be changed to references to the formal parameter, as this now points to the visitor class where the method was originally defined. Finally, recursive calls to the `visit` method of child nodes should be replaced by calls to the `prettyprint` method of these nodes. The following rules can be used to describe these changes:

```
method(?concreteElement,?operationname,[?visitor],?methodid) :-
     ConcreteVisitor(?visitor),ConcreteElement(?concreteElement),
     method(?visitor,?methodname,[?concreteElement],?methodid),
     operationname(?visitor,?operationname).
formalparameter(?methodid,?concreteVisitor,?name) :-
     formalparameter(?methodid,?concreteElement,?name),
```

```
     concreteVisitor(?concreteVisitor),concreteElement(?concreteElement).
thisexpressionnode(?node) :- variableaccessnode(?node,x).
variableaccessnode(?node,x) :- thisexpressionnode(?node).
messagenode(?node,?rec,?operationname,?args)
  :- messagenode(?node,?rec,visit,?args),
     ConcreteVisitor(?visitor),operationname(?visitor,?operationname).
```

These rules make use of the architectural knowledge provided by the developer, as is described in 4.1. Note, for example, how the rules make use of the `operationname` predicate in order to provide a meaningful name to the methods that implement the operation of the visitor. Given these rules and this knowledge, the compiler is able to remove the extra indirections and abstractions introduced by this architecture. The code it emits can then be further optimized by already existing techniques, enabling even better optimization conditions.

6 Future Work

Further investigation into different areas is needed in order to complete the work presented in this paper. First of all, we only tested our approach on some small but prototypical systems, which showed good results. The next step thus consists of trying to optimize real-world object-oriented systems. Second, we will study the impact of our techniques on current optimization techniques. We hold the position that the latter can benefit from our optimizations, as unnecessary abstractions and indirections are removed, which enables better conditions for statically binding and inlining messages. Third, we will integrate our work with the work of [10] and develop a library of transformations in order to automate the optimization process and eliminate developer intervention as much as possible.

7 Conclusion

In this paper, we showed that current techniques for optimizing object-oriented systems fall short when applied to systems which conform to important non-functional requirements such as reusability, adaptability and extendability. This is mainly due to the fact that these techniques only try to statically bind message sends by predicting the exact type of an object at runtime. They do not incorporate global knowledge about the architecture of a system, with all its specific abstractions and relationships between classes and methods. As a consequence, they fail to see that it is the architecture of such systems that is the principal reason for the performance loss. To alleviate this problem, we proposed to use a compiler incorporating architectural knowledge which is able to transform one architecture into another, more efficient one, thereby reducing the total number of messages sent, instead of simply statically binding them. In order to achieve this, a uniform framework was presented, consisting of a representation for a program suited for our purpose, an annotation and a transformation language. Using this framework, developers are able to provide the compiler with architectural information and can specify rules to manipulate and transform the internal representation of the program. We showed how to use this framework for performing architectural optimizations on systems implemented using design patterns.

8 Acknowledgements

The authors would like to thank Theo D'Hondt for promoting the work presented in this paper. Special thanks to Kris De Volder for fruitful discussions about and important contributions to this work. Many thanks to Carine Lucas, Kris De Volder and Kim Mens for proofreading. Thanks to all other members of the Programming Technology Lab for making it an inspiring place to work.

References

1. Kent Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1997.
2. Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, 1992.
3. James O. Coplien. *Advanced C++ programming styles and idioms.* Addison-Wesley Publishing Company, 1992.
4. Kris De Volder. *Type-Oriented Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, 1998.
5. Jeffrey Adgate Dean. *Whole Program Optimization of Object-Oriented Languages.* PhD thesis, University of Washington, 1996.
6. Karel Driesen and Urs Holzle. The direct cost of virtual function calls in c++. In *Proceedings of the OOPSLA 96 Conference*, pages 306–323. ACM Press, 1996.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series, 1995.
8. William G. Griswold. *Program Restructuring as an Aid to Software Maintenance.* PhD thesis, University of Washington, 1991.
9. Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts.* PhD thesis, Vrije Universiteit Brussel, 1997.
10. Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Technology of object-oriented languages and systems*, 1998.
11. Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison Wesley, 1990.