Digging into the Visitor Pattern

Fabian Büttner, Oliver Radfelder, Arne Lindow, Martin Gogolla
University of Bremen, Computer Science Department
E-mail: {green,radfelde,lindow,gogolla}@tzi.de

Abstract

In this paper we present an alternative to the VISITOR *pattern,* DYNAMIC DISPATCHER*, that can be applied to extend existing software in a nonintrusive way, and which simulates covariant overriding of visit methods. It allows to express polymorphic operations through visitor classes in a more natural way than the original* VISITOR *pattern. Our solution* DYNAMIC DISPATCHER *can be applied without touching existing domain classes. Therefore, it is especially useful to extend frameworks and libraries. We have implemented* DYNAMIC DISPATCHER *as a small framework in Java and conducted performance measurements which show that the overhead should be acceptable in many real world scenarios.*

1. Introduction

In the area of software development, agile methodologies like Extreme Programming [3], the Unified Process [12], and others have evolved and lots of projects are done based thereon. These approaches consider analysis, design, implementation, and testing as parallel activities. As a consequence, software design artifacts change continuously. This effect is intended to achieve a design which is constantly adequate to the emerging problem domain.

Design patterns [11] are instruments to obtain a robust, maintainable, and extensible design. One general intention of design patterns is to decouple individual concerns, so that as many parts as possible of a design remain stable according to requirement changes.

One of the most controversly discussed patterns is the VISITOR pattern. The general intention of the pattern is to allow defining operations separated from the classes they operate on. Several problematic properties of VISITOR are mentioned in the literature, for example in [13, 15]. We will target two problems especially: First, applying the pattern to existing software is very intrusive. Thus, it often cannot be used to extend existing software, particularly frameworks, as discussed in [19]. Second, in mainstream object-oriented programming languages like Java, C++, C#, and others that do only support invariant method overriding [4, 7], VISITOR cannot express specialization. Therefore, VISITOR is not able to extract inherited operations into Visitor[1] classes.

In this paper we present an alternative to the VISITOR pattern, DYNAMIC DISPATCHER, that can be applied to extend existing software in a nonintrusive way, and which simulates covariant overriding of *visit* methods. It allows developers to express polymorphic operations through visitor classes in a more natural way than the original VISITOR pattern. DYNAMIC DISPATCHER can be applied without touching existing domain classes. Therefore, it is especially useful to extend frameworks and libraries.

To demonstrate that it is easily realizable, we have implemented DYNAMIC DISPATCHER as a small framework in Java using several strategies. Since we achieve the dynamic dispatching by executing additional code, there is the danger of a substantial performance impact. We conducted performance measurements which show that the relative overhead should be acceptable in many real world scenarios.

This paper is structured as follows: Section 2 introduces a simple design example, on which we apply the VISITOR pattern. In Sect. 3, we discuss the problems of VISITOR that motivate our variant DYNAMIC DISPATCHER, which is explained in Sect. 4. In Sect. 5, we present how DYNAMIC DISPATCHER can be implemented as a framework in Java. Section 6 shows the results of the performance measurements we conducted with this framework. Finally, in Sect. 7, we make a conclusion and present future work.

2. Graphics Example

In the following we will present the design of a simplified, hypothetical vector graphics software on which we will discuss some problems of VISITOR. The UML class diagram in Fig. 1 shows the static structure of the example: a picture can be composed of shapes, which can be lines, arrows, and other pictures. The reader may notice this reflexive relationship as the COMPOSITE design pattern. Two

[1] We use the following conventions: 'VISITOR' refers to the pattern, 'Visitor' refers to the class in the pattern, and 'visitor' refers to the general concept.

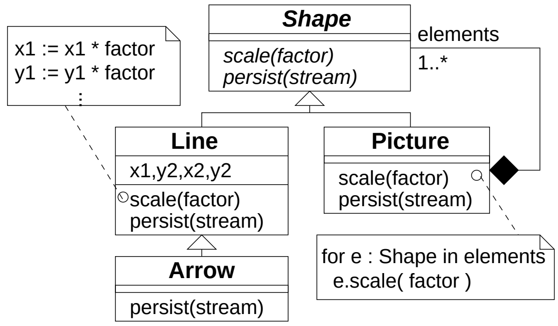

Figure 1. Graphics example

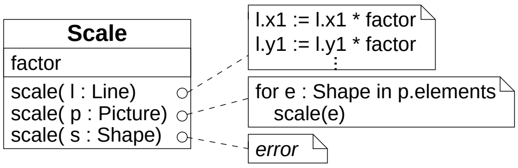

Figure 2. Extracted operation (not working)

operations are defined for shapes: *scale* and *persist*. Both operations are recursive as they descend into the tree of elements formed by the composition. For example, *scale* is applied to all elements within a picture, which may be pictures themselves, and so on.

Both shape operations are redefined (overridden) in the child classes Line and Picture. While class Arrow overrides *persist* it does inherit *scale*. Therefore, the implementation language is required to late-bind method invocation to correctly call *scale* and *persist* for lines, arrows, and pictures. Java, C++, C#, and most other static typed OO programming languages support this kind of method binding[2].

This is common object-oriented programming style. However, in certain situations the software developer may want to separate behavior from state, i.e., to define (some) operations outside the domain classes Shape, Line, Arrow, and Picture. Although this conflicts with common understanding of object-oriented programming in a certain sense, there are at least two recurring motivations in software engineering that justify this violation: One is to keep domain classes independent from operations in order to enable their reuse and to keep them stable when operations change. The other motivation is the use of libraries or frameworks. Since libraries typically cannot be changed, there is no alternative to defining new operations some other place. In general, this place is another class.

For example, in our graphics software the *scale* operation could be separated from our domain classes into a class Scale, as depicted in Fig. 2. The class Scale consists of three

[2]We use 'method' to refer to an implementation, otherwise we use 'operation'

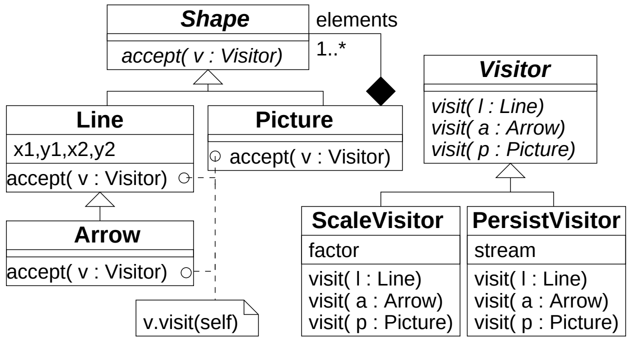

Figure 3. Graphics example - VISITOR applied

scale methods, one for each occurrence of *scale* in the domain classes before. One needs an instance of this class to scale shapes.

Unfortunately, this code does not work in mainstream object-oriented programming languages. The method *scale(l:Line)* is never called from within *scale(p:Picture)*. Instead, *scale(s:Shape)* is executed regardless of the runtime type of the elements in the picture. That is because these languages do not support double-dispatch of method invocations. Instead, the method to be invoked is chosen solely based on the runtime type of the receiver (self). If we had omitted *scale(s:Shape)*, Scale would not even compile. *scale* is only overloaded and the call *scale(e)* is statically bound to a method according to the argument's reference type. An in-depth comparison of the different late-binding signatures of Java, C++, C# and others can be found in [4].

A common solution to this problem is the VISITOR pattern. The basic idea is to call an (invariantly) *overridden* method *accept* in the class hierarchy you work on. In turn, this method calls the correct *visit* method (as *persist* is now named) on the caller. Figure 3 shows the static structure of our graphics example after applying VISITOR. The *accept* method always consists of the same piece of code: *visit(self)*. Consequently, the early-bound call to the overloaded method *persist* is replaced by a late-bound call to the overridden method *accept*. The type of *self* is always the type of the class, therefore it is always different and the correct overloaded method is selected in the caller at compile-time. Some people append the type name to Visitor methods, i.e. *visitline*, *visitarrow*, and so on to make this fact clearer, but there is no difference at this point between overloading and renaming. Since nothing special happens in the *accept* method, we can reuse it for all operations we want to move out of the class hierarchy, like *scale*, *persist*, etc. The classes that implement these operations must adhere to the contract that they understand a *visit* call for each concrete class in the hierarchy. This is achieved through a common abstract base class (or an interface) *Visitor*. Consequently, the functionality of *scale* must be included in both

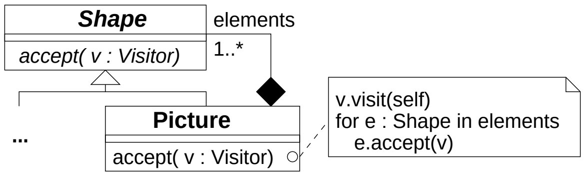

Figure 4. Traversal encoded in the objects

ScaleVisitor::visit(e:Line) and *ScaleVisitor::visit(e:Arrow)*. We will discuss this issue in Sect. 3.2.

Considering the dependencies between shapes and operations, we notice that shapes are now *decoupled* from subclasses of Visitor in the way that concrete Visitors (ScaleVisitor, PersistVisitor) are not known to concrete Shapes at compile-time. As an effect, we may change or add new operations on shapes without recompiling Shape, Line, Arrow, or Picture. Another side-effect regards attribute and method visibility: Since the *scale* methods now reside in Scale, they must have either access to the attributes in Line and Picture, or there must be some state-exposing interface in these classes. It is evident that applying VISITOR may break encapsulation, since at least the visitors need access to the object states.

A common variant of the VISITOR pattern is to leave the *traversal* through the object structure in the *accept* methods (Fig. 4). If all algorithms use the same way to iterate through the objects, redundant traversal code can be avoided. Perhaps even more important, the kind of composition can be left private, weakening the visitor's dependency. On the other hand, future visitors are restricted to one kind of traversal. Thus, it seems to us, that this variant of the VISITOR pattern requires a lot of foresight, and should be applied carefully. Other design patterns may be used instead to avoid duplicated traversal code in the visitors (e.g. STRATEGY, ITERATOR [11]).

3. Problems Induced by VISITOR

Apart from problems that are intrinsic to the general idea behind VISITOR, like breaking encapsulation and introducing a level of indirection, there are several other problems that are solved in certain approaches. We address three of those problems with our DYNAMIC DISPATCHER.

3.1. Visitor is Intrusive

It is obvious that applying VISITOR to extend existing software by either extracting or creating new operations is a very intrusive procedure. Firstly, the domain classes must provide an *accept* method. Furthermore, for inherited methods explicit delegation code must be created due to the problem of having no implementation inheritance (see below). Therefore, VISITOR is definitely a heavy-weight pattern, which may not be applicable to existing software in several cases. This problem is also discussed in [16], which presents a generic 'Walkabout' class to replace *accept*.

Another undesirable effect of VISITOR is the cyclic dependency relationship between classes and subclasses: superclasses know their subclasses, because Visitor knows all domain classes through the parameter types of its *visit* methods, and each domain class knows Visitor. [14] and [15] avoid cyclic dependency in ACYCLIC VISITOR and EXTRINSIC VISITOR.

3.2. No Implementation Inheritance

The VISITOR design pattern does not support implementation inheritance. In a design without a visitor (operations are defined in the classes they operate on), subclasses inherit the methods they do not override. In our example, Arrow inherits *scale* from Line. Nevertheless, ScaleVisitor needs to implement *scale(a:Arrow)* - which may call *scale(l:Line)*. Thus, implementation inheritance must be manually simulated when applying VISITOR in the general case.

Assuming we left out *visit(a:Arrow)* in the Visitor class, arrows would be handled as lines in each subclass of Visitor (i.e. in each extracted operation). As soon as one visitor must differentiate between lines and arrows, all visitors must do so. The reason is *again* the typical OO language's uni-dispatch late-binding: because calls to *visit* are early bound with regard to the argument type, *visit* can not be specialized in a visitor. If our language supported covariant method overriding (or multimethods) like Dylan [2] and CLOS [9], this kind of specialization would be possible. Manually simulating implementation inheritance can become time-consuming, hard to maintain, and error prone: If one Visitor needs a special method for a certain domain class not yet present in the Visitor base class, all other visitors must be changed as well. Therefore, as discussed in [19], it is not possible to use VISITOR in frameworks/libraries, because the framework user cannot change the interface of the visitor base class, even if she is allowed to introduce new domain subclasses. In all cases, changes to the inheritance relationship between domain classes must be reflected in the manually coded delegation, with the danger of introducing hardly traceable errors.

One solution to overcome these problems is DEFAULT VISITOR [15], which introduces a common base class (DefaultVisitor) for visitors to inherit from. In DefaultVisitor, each *visit* method calls the next more general method, up to the most general argument type. Thus, individual visitors only need to override some *visit* methods. Still, DEFAULT VISITOR is not applicable to frameworks.

Another solution is to move the dispatching to the correct *visit* method into its own method (*dispatch*) in the visitor. This method checks the runtime type of its argument and calls the most specific *visit* method. This way, implementation inheritance is simulated, and additional Shape classes

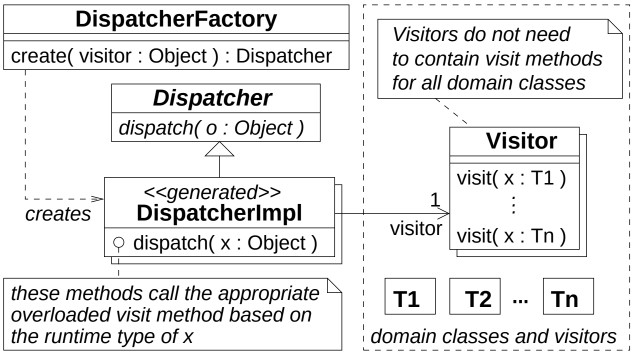

Figure 5. Structure of DYNAMIC DISPATCHER

are not intrused with *accept* methods. The major drawback of this approach is that we must manually maintain several *dispatch* methods (e.g. one for Scale and another for Persist). This solution is called EXTRINSIC VISITOR by [15].

4. Dynamic Dispatcher

Reconsidering how we motivated applying VISITOR, we recall that we tried to separate polymorphic operations from the domain objects they work on. VISITOR was motivated by the fact, that Java and most other common object-oriented programming languages only support uni-dispatch late-binding. If we had a language which supports at least double-dispatch method binding, it would not be necessary at all. Especially for Java, several approaches were discussed to introduce double-dispatch method binding (e.g. [10]). Most approaches we know require a modified compiler, or a modified virtual machine, they may not be adoptable for many applications. One exception is the Java Multi-Method Framework [18], which covers general multi-methods via reflection, but introduces a significant performance overhead to normal method invocation.

In the following, we present another solution DYNAMIC DISPATCHER, which specially targets the cases where VISITOR may be applied, and which does not require any changes to the compiler, virtual machine, or runtime. At its heart, we introduce a *Dispatcher* object, that dynamically chooses the most appropriate *visit* method. Thus, DYNAMIC DISPATCHER replaces the *accept* methods of the VISITOR pattern by an explicit dispatching object. This object is generated at runtime by passing a visitor object to a factory that dynamically derives the dispatcher object.

4.1. Structure

The general structure is depicted in Fig. 5. The classes Dispatcher and DispatcherFactory are part of the dispatching framework, while Visitors are arbitrary user defined classes that declare some *visit* methods. There is no abstract

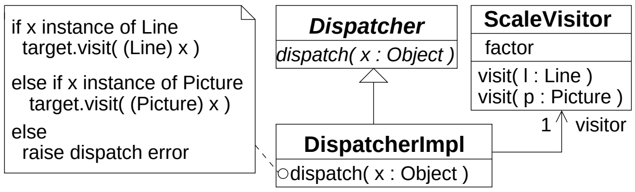

Figure 6. Dynamic dispatcher for Scale

base class Visitor, as in the original pattern, which requires a fixed set of *visit* methods to be defined.

A DispatcherImpl object is generated by *DispatcherFactory::create* and holds a reference to its visitor. The dispatching, i.e. calling the appropriate *visit* method, is done in the overridden *dispatch* method. Notice, that although not shown here, a Visitor may need a reference to the Dispatcher to (recursively) invoke its own *visit* methods, like in traversal operations. Although not strictly necessary, the DispatcherImpl class should be typically generated at runtime, as explained below.

4.2. Choosing Appropriate Visit

What does choosing the appropriate *visit* method for an argument *x* of *dispatch* mean? Basically, if there is more than one *visit* method with an argument type to which *x* is assignable, the one with the most specific argument type should be called. In general, in programming languages with subtyping (see [6, 1]), there may be no most specific type. This is the case if none of the assignable argument types is a subtype of all others. If there is no most specific type for *x*, then there is no most appropriate *visit* method. We regard this situation as an ambiguity error.

For Java and C#, we can avoid this ambiguity by restricting the allowed argument types in the *visit* methods to class types. Because Java and C# prohibit multiple inheritance between classes, this restriction ensures that if there is at least one compatible *visit* method for *x*, there is always a most specific method. Less restrictive solutions exist (e.g., [5],[18]), but they are more complicated. However, if our intention was to move operations out of a class hierarchy, we do not impose any restrictions, because the only place a method can be defined in Java is a class.

Let us explain a dispatcher generated for a ScaleVisitor that works on our shape-hierarchy, as depicted in Fig. 6. The visitor consists of *visit* methods for Line and Picture. The *dispatch* method sequentially checks the parameter against the types of the *visit* methods, and calls the appropriate one. Notice, that we gained ‘implementation inheritance’. In the previous section, we have defined Arrow as a subclass of Line, so that the expression *x instance of Line* yields true for an instance *x* of Arrow. Hence, passing an Arrow object to *dispatch* results in the execution of

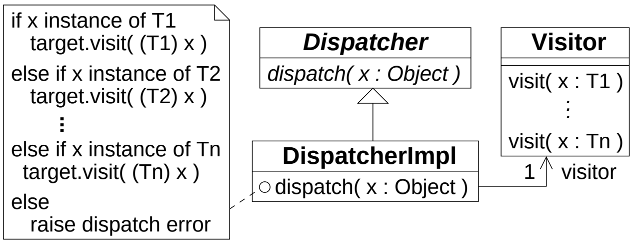

Figure 7. Simple dispatch algorithm

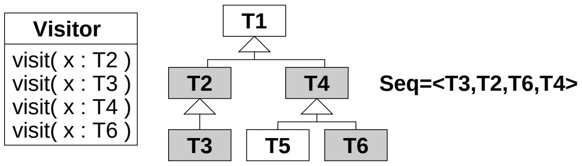

Figure 8. Deriving a sequence for dispatch

visit(l:Line). This corresponds with our intention that scaling arrows is inherited from scaling lines.

In contrast, persisting shapes differentiates between arrows and lines, expressed by a separate *visit(a:Arrow)* method in PersistVisitor. Therefore, the corresponding dispatcher must also check against class Arrow. Actually, this check must happen before the one against Line, otherwise *visit(l:Line)* would be called unintendedly.

In general, the *dispatch* method can be implemented as a sequence of *if*-statements. The inheritance tree between classes (without multiple inheritance) can be transformed into a sequence $T_1, \ldots, T_n$, so that the *dispatch* method looks like Fig. 7. For $T_1, \ldots, T_n$, $(T_i$ *subtype of* $T_j) \Rightarrow i < j$ must hold to ensure that more specific types are checked first.

Figure 8 illustrates how to derive such a sequence from a set of types. The grey-shaded boxes in the class diagram are the classes occurring as argument types of the *visit* methods. The depicted sequence of types is one of the four valid sequences for the simple dispatch algorithm used in Fig. 7.

One may argue that the implementation as a sequence of *if*-statements is against the spirit of object-orientation. After all, the removal of conditional statements is one major advantage of OO. Yet, *dispatch* enables us to express visitor objects in a far more natural way. We gain simplicity in many other places by violating the OO paradigm in one place. Furthermore, as shown later, the dispatcher implementation can easily be generated on demand at runtime, removing the drawback of manually coded *if*-statements.

Other implementation techniques of *dispatch* are possible. For deep class hierarchies, it may be more efficient to reorder the *if*-statements to perform a tree search along the class hierarchy to find the appropriate *visit* method. More complex, already found method resolutions may be cached (e.g. in a hash table), or precomputed (e.g. as in [17]). Unfortunately, the presence of dynamic linking and multi-threading requires synchronization. We experienced that the simple linear search algorithm is sufficient in many cases (see Sect. 5).

4.3. Consequences and Requirements

In Sect. 3 we discussed three major problems of the VISITOR pattern: intrusiveness, cyclic dependencies, and the lack of implementation inheritance. DYNAMIC DISPATCHER addresses these problems.

Our variant allows to define new functionality over domain classes in the same way VISITOR does. In contrast to VISITOR, it does not require changes to the domain classes. Thus, it is not intrusive and can be applied to extend existing frameworks and libraries. As a consequence of the fact that domain classes do not need an *accept* method in our approach, DYNAMIC DISPATCHER does not introduce cyclic dependencies between domain classes.

Finally, we simulate covariant overriding of *visit* methods. Therefore, DYNAMIC DISPATCHER allows to simulate implementation inheritance in visitor operations. This means that developers who add visitor operations to domain classes can *specialize* them for individual domain classes in the same natural way they do when they simply override methods in the domain class hierarchy. Consequently, DYNAMIC DISPATCHER based implementations are better maintainable and more robust against future changes to domain classes as well as changes to individual visitors. Also, it allows developers to express functionality clearer and more concise than VISITOR, because no dispatching code clutters the visitor class.

In languages with dynamic linking of types, the dispatching algorithm cannot be derived at compile time. At least, we need some kind of *runtime type reflection* mechanism to analyze the method signatures of a certain visitor class. That is, which *visit* methods are available and which are more special than others. We also need a way to determine if an object is an instance of a certain type to find the most appropriate method.

The actual dispatching of an invocation can be implemented via reflection, if dynamic method invocation is supported. Alternatively, dispatching code can be generated on the fly. We present a small framework in Sect. 5, which we used to evaluate both dispatching variants for Java.

There are two weak points in comparison to VISITOR: The first is, DYNAMIC DISPATCHER may decrease performance due to the additional dispatch code to be performed. We will discuss some performance measurements we conducted in Sect. 6. The second is, DYNAMIC DISPATCHER performs type checking at runtime. If no appropriate *visit* method is found, some kind of dispatch error must be raised. Therefore, in DYNAMIC DISPATCHER type errors can occur at runtime, whereas they are detected by the compiler in the VISITOR pattern.

5. Implementation

For evaluation purposes, we have developed a small framework that realizes DYNAMIC DISPATCHER in Java. In the following, we give a brief overview of the implementation and discuss some performance results.

We have realized three different dispatcher factories. One can choose each of these to create an instance of Dispatcher for a particular object that contains *visit* methods.

Our first solution (SCDispFactory) is straightforward: *SCDispFactory::create* analyzes the given visitor object for *visit* methods. The argument types are extracted, and ordered as depicted in Fig. 8. Then, source code for a Java class that implements dispatch as in Fig. 6 is written to a temporary text file and compiled at runtime by the Java Compiler interface. The resulting class is loaded through a custom class loader and finally an instance of it is and returned.

The second solution (ReflectiveDispFactory) reuses a generic class that is initialized with the type sequence as described above. Whenever *dispatch* is called on this object it iterates over the sequence until the first assignable type is found and invokes the corresponding method by reflection.

Both solutions suffer from several problems which we will discuss later on. Our last dispatcher factory (BCDispFactory) avoids these problems. It works similar to the source code generator, but instead of compiling the dispatcher class from a temporary text file it directly defines the class in byte code. The byte code is generated using the free Bytecode Engineering Library (BCEL) [8] which provides helper classes for writing Java byte code.

6. Performance Analyis

To estimate the impact of our DYNAMIC DISPATCHER implementation, we have conducted two performance suites where we compared the three aforementioned dispatcher factories with the original visitor, and of course with the initial, object-oriented form. We will discuss the results at the end of this section. All tests were executed on a single user, 1.2 GHz Pentium III machine running Windows XP and J2SDK, Version 1.4.2.

6.1. Test Suites and Results

The first suite ('raw') is meant to estimate the cost of a single method dispatch. Because our three dispatcher implementations all perform a linear search to find the appropriate method, we parameterized this suite with the number of classes n. For a single run we create n subclasses C_i of a Base class, each overriding the Base methods *f* and *accept*. For VISITOR and DISPATCHER, we measure the time

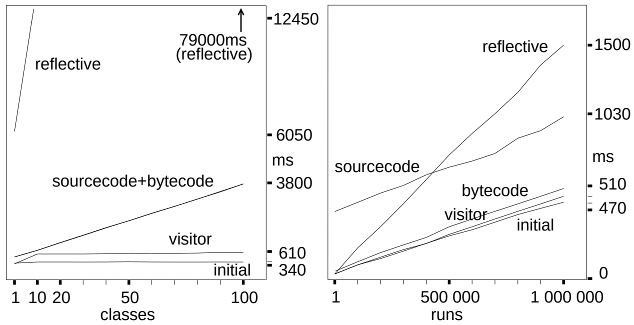

Figure 9. Performance comparison

to invoke *visit* through *accept* resp. *dispatch* with randomly chosen C_i instances as arguments. For the initial form, we measure the time of a simple call of *f*. To eliminate the effects of test setup, we perform a large number of dispatches (ten millions) and show the total time. In our simplistic linear search approach it makes no difference whether the class hierarchy is flat or deep.

The left-hand side in Fig. 9 shows the results of this suite: the initial form, which is a simple method invocation of *f*, takes a constant amount of time. As estimated, the execution time of source and byte code generated dispatchers is nearly equal, as they only differ in the way they are generated. They grow in a linear way with the number of classes n. For $n = 1$ they take twice as much time to execute as the initial form, with $n = 100$ they take ten times the amount. The reflective dispatcher is much slower: its ratio grows from 18 at $n = 1$ to 230 at $n = 100$. The visitor takes always about twice the amount of the initial form, except for $n = 1$, where they are equal. We wondered about this point and found out, that this break out vanishes, when Java's just-in-time compiler is deactivated.

While the first suite measures the raw cost of method invocations, the second one ('scale') measures the overall effect of using a dispatcher in a complete algorithm. We have implemented the Scale algorithm in Fig. 1 and 2 in all forms (initial, visitor, byte code, source code, and reflective dispatcher). Then we scaled a simple picture, which consists of arrows and lines multiple times, and measured the overall execution time. In contrast to the first suite, we included the time required to construct the dispatcher and visitor objects.

The result is shown in the right-hand side of Fig. 9. As expected, all graphs increase in a linear way. The reflective dispatcher is by far the slowest while the others perform nearly equal. After one million runs, the difference between byte code dispatcher and the initial form is less than ten percent. The source code generated dispatcher increases by the same degree as the byte code generated dispatcher, but takes a quite large amount of time to construct the dispatcher class.

6.2. Performance Consequences

Our test suites are not exhaustive, but they give some hints about how expensive our approach is. Method invocation through the byte code generated dispatcher is slower than normal method invocation and invocation through an *accept* method. However, the relative overhead decreases significantly if some (even little) code is executed within the invoked method. To provide a number, the overhead in our graphics example is about ten percent. We expect that this overhead is negligible in many real world scenarios.

If only few calls are to be performed, the reflective dispatcher may also be sufficient. Its advantage is that it can be implemented as a single generic reusable class.

7. Conclusion and Future Work

We presented a variant of the VISITOR pattern that is more suitable for framework designs and more natural for software developers in certain situations. It can be applied to extend software without affecting existing code. We understand it as another tool that is especially useful in agile software development.

We demonstrated how DYNAMIC DISPATCHER can be implemented as a small framework in Java. We could have done this in C# and the .NET framework as well, since .NET also provides the required reflection capabilities. For C++, at least the sketched generated source code dispatcher can be implemented. Our evaluation showed that the performance tradeoff implied by the implementation should be acceptable in many real world situations.

One of the drawbacks of DYNAMIC DISPATCHER is that type errors can occur at runtime. We are currently working on an extension to achieve more static type checking. Basically, we are going to allow to specify a user defined base type for the dispatcher object.

Apart from these technical aspects we are going to further evaluate the applicability of DYNAMIC DISPATCHER for a larger project in which the aforementioned problems of VISITOR arise. We are going to realize the project with both variants. We expect that the DYNAMIC DISPATCHER solution will take less time to be completed and results in a simpler overall design.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, New York, 1996.

[2] Apple Computer, Eastern Research and Technology. *Dylan: an object-oriented dynamic language*, 1992.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[4] A. Beugnard. OO languages late-binding signature. Ninth International Workshop on Foundations of Object-Oriented Languages, 2002.

[5] J. Boyland and G. Castagna. Parasitic methods: Implementation of multi-methods for Java. In *Conference Proceedings of OOPSLA '97, Atlanta*, volume 32(10) of *ACM SIGPLAN Notices*, pages 66–76, New York, NY, 1997. ACM.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[7] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[8] M. Dahm. Byte code engineering with the BCEL API. In C. H. Cap, editor, *Java Informationstage '99*, Informatik aktuell, pages 267–277, 1999.

[9] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In J. Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 151–170. Springer, New York, NY, 1987. LNCS, Volume 276.

[10] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi-Dispatch in the java virtual machine: Design and implementation. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-01)*, pages 77–92, 2001.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object- Oriented Software*. Addison-Wesley, 1997.

[12] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, 2001.

[13] R. C. Martin. Acyclic visitor. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 93–104. Addison-Wesley Publishing Co., Reading, MA, 1998.

[14] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Pearson Education, 2002.

[15] M. E. Nordberg. Default and extrinsic visitor. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 105–123. Addison-Wesley Publishing Co., Reading, MA, 1998.

[16] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 1998.

[17] C. Pang, W. Holst, Y. Leontiev, and D. Szafron. Multimethod dispatch using multiple row displacement. *LNCS*, 1628:304–329, 1999.

[18] Remi, F. Etienne, and D. Gilles. Java multimethod framework. *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, 2000.

[19] J. Vlissides. Visitor in frameworks. *C++ Report*, 11(10):40–46, 1999.