On the Efficiency of Design Patterns Implemented in C# 3.0

Judith Bishop[1] and R. Nigel Horspool[2]

[1] Department of Computer Science, University of Pretoria, Pretoria, South Africa
jbishop@cs.up.ac.za http://www.cs.up.ac.za/~jbishop
[2] Department of Computer Science, University of Victoria, Victoria, BC, Canada
nigelh@cs.uvic.ca http://www.cs.uvic.ca/~nigelh

Abstract. From the very inception of design patterns, there was the anticipation that some of them would be superceded by new language features. Yet published implementations of classic patterns do not generally live up to this promise. The occurrence of generics, delegates, nested classes, reflection and built-in iteration is confined to a few patterns in a few published compendiums in a few languages. In this paper we trace the interplay between languages and patterns over the past decade and investigate how relevant language features really are for pattern implementation. We back our conclusions with a detailed look at the visitor pattern, examining the impact of C# developed in the past few years. We conclude that efficiency should play a large role in the choice of design pattern implementation, since some new features still bring with them runtime overheads.

Key words: design patterns, efficiency, C# 3.0, visitor pattern, reflection, dynamic dispatch, delegates

1 Introduction

Design patterns represent a layer of abstraction above that of programming language features. They are rapidly becoming a unit of discourse among software developers, and have transformed the way in which software is designed and discussed. The distance between design patterns and programming language features is, however, by no means constant. It varies between different languages, and more particularly within the same language as new features are introduced. Over the years, language design has evolved so that some patterns can be implemented almost directly. For example, the interface mechanism common in Java and C# almost dissolves the Template Method pattern. Not all patterns have benefited equally. Most implementations of the Visitor pattern still follow a complex structure of class and method interactions, made easier but not completely transparent, by advanced features such as generics and reflection.

These tensions are summarized in the diagram in Figure 1. We start with the full set of 23 patterns [11]. If a new language feature is included in an implementation of a pattern, it might have an effect on efficiency — for better or worse.

It is this drop in efficiency that we wish to explore and quantify. An exhaustive study of all the patterns is underway: this paper is a starting point. It surveys and brings up to date the research on this issue over the past decade; it highlights certain patterns that are known to be responsive to language changes, and presents the results of experiments for one of the most complex patterns, the Visitor. An important contribution of the paper is the survey of methodologies used so far to match language features to patterns, thus enabling those in the higher efficiency group to be identified.

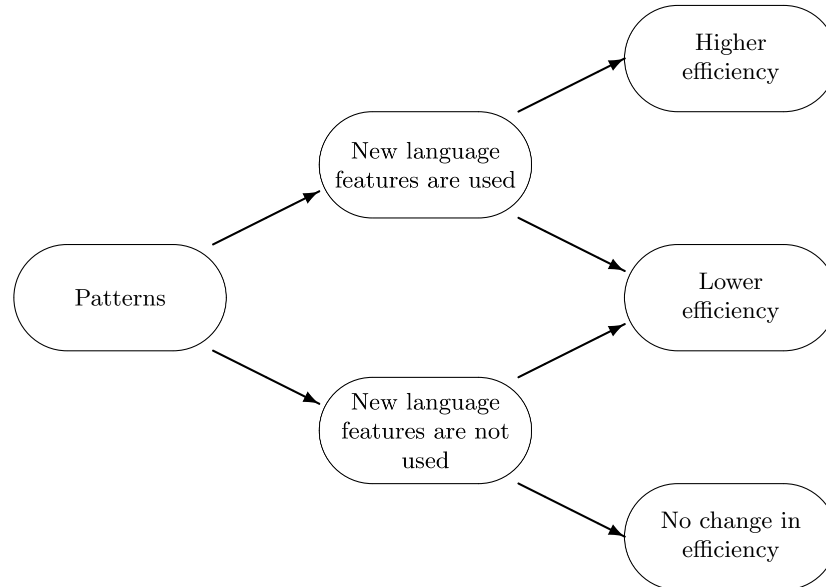

Fig. 1. The effect of language evolution on design pattern efficiency

2 Relationship between patterns and languages

The abstraction level of languages rises constantly [4], having an inevitable effect on the way in which patterns are implemented. There is a movement towards leveraging the value of patterns by componentizing them in libraries. The background to these two movements, and their convergence, is now discussed.

2.1 Replacing patterns by features

Design patterns provide a high-level language of discourse for programmers to describe their systems and to discuss common problems and solutions. This language comprises the names of recognizable patterns and their elements. The

proper and intelligent use of patterns guides developers into designing systems that conform to well-established prior practices, without stifling innovation.

The original group of patterns was implemented in the era of C++, with some tempering by Smalltalk [11] and reflected the capabilities of those languages, for good and bad. Indeed, Gamma states that

> If we assumed procedural languages, we might have included design patterns called "inheritance", "Encapsulation", and "Polymorphism". Similarly, some of our patterns are supported directly by less common object-oriented languages.

It was certainly not an aim of design patterns to force a certain way of coding, thus deprecating the value of new language features that could make design patterns significantly easier to express.

The debate over which language features these would be began a decade ago [3, 12, 1, 6, 7]. Java and more notably C# have added significant language features over the last decade. For example, C# 2.0, which was developed between 2002 and 2005, added generics, anonymous methods, iterators, partial and nullable types. C# 3.0, finalized in 2007, focuses on features that would bring the language closer to the data that pours out of databases, enabling its structure to be described and checked more accurately. These features included: implicit typing of local variables and arrays, anonymous types, object and array initializers, extension methods, lambda expressions and query expressions (LINQ).

Two studies that have looked at how to identify the patterns that would benefit from new language features, and thus would fall in the upper right part of Figure 1. Agerbo and Cornils [1] list reasons why it is important to divide patterns into the following classes:

- Language Dependent Design Patterns (LDDPs)
- Related Design Patterns (RDPs)
- Fundamental Design Patterns (FDPs)
- Library Design Patterns (LDPs)

The first category consists of patterns that are covered by a feature in one language, but not yet all. For example, the Factory Method aims to create objects whose exact classes are unknown until runtime. Using interfaces, which are common these days, the creator class can be bound at runtime. Similarly, the Chain of Responsibility pattern can make use of the delegate feature in languages such as C# and VB. Agerbo envisages that once the features supporting a pattern in the LDDP group become widely available across languages, the pattern essentially loses its status. It can then be removed from the pattern language of discourse. An advantage of this approach is that the number of patterns will stay within manageable proportions. The first column of Table 1 lists the patterns that Agerbo regarded as LDDPs.

RDPs are those that can be implemented using another pattern. The example quoted is the Observer pattern that can be implemented using the Mediator. Another common relationship is the Interpreter pattern which uses the Visitor [5]. The patterns that remain are then the fundamental ones, and according to

	LDDPs	***Cliches and idioms***
Chain of Responsibility	Delegates	
Command	Procedure classes	classes
Fa cade	Nested classes	encapsulation
Factory Method	Virtual classes	
Memento		persistence
Prototype	Pattern variables	Deep copy
Singleton	Singular objects	module
Template method	Complete block structure	overriding
Visitor	Multiple dispatch	Multi-methods

Table 1. Patterns supported by language features, the LDDPs of Agerbo and Cornils [1] and the cliches/idioms of Gil and Lorenz [12]

this study they comprise the 11 shown in Table 2. Of course, we can immediately see that the list is dated, because Iterator and Memento, for example, have been covered by advances in both Java and C# (iterators and serializable respectively). Nevertheless, the classification is a useful one because it can be applied to the burgeoning group of new patterns, not just to the original 23. We come back to Agerbo's LDPs under the Componentization section.

	FDPs	***cadets***
Bridge	X	X
Builder	X	X
Composite	X	X
Decorator	X	X
Mediator	X	X
Proxy	X	X
State	X	X
Adapter		X
Chain of Responsibility		X
Interpreter		X
Observer		X
Strategy		X
Visitor		X
Abstract factory	X	
Flyweight	X	
Iterator	X	
Memento	X	
Total	11	13

Table 2. Fundamental patterns identified as FDPs by Agerbo and Cornils [1] and as cadets by Gil and Lorenz [12]

Gil and Lorenz [12] earlier attempted a similar classification. Their cliches and idioms could mimic features found in languages, whereas cadets are still candidates for language support. Their standpoint was that patterns, once identified as such, could and indeed should grow into language features. This group is shown in the second column of Table 1. At the time, the ones they identified as still requiring support are shown in Table 2.

Assessment. Both papers put forward the belief that the ultimate goal is that the patterns left in Table 2 should eventually develop into single language features or programming language paradigms. We do not fully agree with this standpoint for these reasons:

1. A feature is much more broadly applicable than just for one pattern. Delegates, for example, are used in Adapters, Mediators and Observers. We would be losing information if all these solutions were subverted under the term “delegate”.
2. A pattern needs more than one prominent feature. The Visitor is an example here: the complexity of its design cannot be replaced by one single feature.
3. Patterns can have valid alternative implementations. The choice would be based on non-functional properties such as maintainability, traceability and most of all efficiency. One might be easier to understand, the other more efficient.

Thus every great programming idea cannot be turned into a realistic language feature, otherwise languages would explode. There will always be a place for design patterns.

Other work. In an in-depth but unpublished report that predates the above two, Baumgartner, Laüfer and Russo [3] make a call for features that will contribute to the implementation of design patterns, including (in modern parlance) interfaces, singleton variables, generics, metaclass objects, and multiple dispatch. In the ensuing years, language design has either realized the suggestions or has moved in other directions.

Bosch’s work [6] concentrates on the reasons why languages should mimic design patterns, and gives several detailed examples in the context of a layered object model. His four point check list — traceability, the self problem, reusability and implementation overhead — is still valid today, but the contribution of his solutions is dimmed by being couched in terms of an object modeling language LayOM. The solutions presented address the first three points, but little mention is made of improvements in implementation overhead.

Subsequently, Chambers [7] examined the influence of the experimental languages of the day (Cecil, Dylan and Self) and looked ahead to first-class generic functions, multiple dispatching and a flexible polymorphic type system, all of which are once again in mainstream languages now.

2.2 Libraries of design patterns

The traceability problem mentioned by [6] can be addressed by implementing patterns as library components. When using such an LDP [1] it will be possible to trace from which design pattern the implementation ideas come. Arnout and Meyer [15] are keener on the benefits to be reaped by reuse of pattern implementations. Having analyzed the standard patterns, they conclude that fully two-thirds admit of a componentized replacement, enabling developers to rely on an API from a pattern library rather than re-implementing the pattern. However, achieving this level of re-use relies on some high level language features present in Eiffel, including genericity, agents, tuples, design by contract and multiple inheritance. Agerbo [1] reports being able to make library routines for 10 of the 23 patterns (43%) using, in particular, generics, virtual classes and nested classes.

There is some disagreement over the efficiency of this approach. Very early on, Frick *et al.* [9] highlighted the tension that exists between flexibility of a robust library and the efficient implementation of a class or method. Agerbo [1] reports that applying a design pattern from a library reduces the implementation overhead, whereas Arnout and Meyer present mixed results: the Abstract Factory pattern is report as having no overhead in the library implementation [2] but the Visitor pattern suffers a 40% degradation [15]. However, the Visitor pattern has long been known as sensitive to tinkering [17] and the library orientation might not be the only cause for the drop in performance. (We return to this theme later.) Another study [10] looks at a library version of the Strategy pattern, specifically aimed at embedded code. The concern is that virtual method calls that are key to this pattern really slow down execution. The solution proposed is to generate compile-time generated and optimized code, and the results show an improvement in both code size and speed.

The major effort in library support has now switched to the use of aspects with design patterns. Hannemann and Kiczales [14] were able to make routines for 12 patterns (52%). The paper presents results for the locality, reusability, composition transparency and (un) pluggability of the AspectJ versions of patterns. The benefits of localising patterns as aspects are the inherent code comprehensibility, better documentation and the potential for composability. It is worthwhile noting that these are very much the same benefits that are claimed for using higher-level language features.

3 The impact of C# 3.0

The reader will have noticed that the majority of references in the previous section are to studies done in the late nineties (the exception being the work of Arnout and Meyer with Eiffel [2]), when C++, Java and the usual host of experimental languages were the prime contenders for pattern implementation. We choose to examine the impact of features in C# 3.0, a commercial object-oriented language, 10 years on.

3.1 Features in C#

C# 1.0, announced with the first .NET release in 2000 made significant advances over Java, shown in Table 3. C# 2.0 added five important features in 2005, especially generics, which had been available in some implementations for two years. C# 3.0, finalized in 2006, focused on features that would bring the language closer to the needs of databases and has a distinct functional feel about it [16].

C# 1.0 (2002)	C# 2.0 (2005)	C# 3.0 (2007)
structs properties foreach loops autoboxing delegates and events indexers operator overloading enumerated types with IO in, out and ref parameters formatted output	generics anonymous methods iterators partial types nullable types generic delegates	implicit typing anonymous types object and array initializers extension methods lambda expressions query expressions (LINQ)
API Serializable Reflection	standard generic delegates	

Table 3. The development of C#

It is still open territory as to whether, and how, these new language features should be used in implementing design patterns. In books and writings on web sites the pull of custom is very strong. Because implementations of the patterns were originally given in C++ and Smalltalk, which have their own particular object-oriented styles, the translations into other languages have not always been completely satisfactory. It is a challenge to make the most of a language, while at the same time retaining the link with the design pattern and its terminology. Although design patterns do not force a certain way of coding, a look at the expository examples in most Java or C# books will show little deviation from the C++ style of the 1990s. It would seem that the promise of language features making patterns easier to implement has been slow to realize. The features are there now, and it is a question of showing how they can be used, and in assessing their efficiency. Not all the features listed in Table 3 are directly relevant for patterns, but a surprising number are.

3.2 Pattern implementation in C#

We now present the results of two complete implementations of the patterns, from DoFactory [8] and Bishop [5].

DoFactory is a commercial organization that sells frameworks of patterns in C# and Visual Basic. They are widely consulted. Each pattern comes in three versions, known as Structural (not to be confused with structural as a pattern group), RealWorld and NETOptimized. The first version usually follows a direct implementation of the classic UML diagrams from Gamma *et al.* [11]. The RealWorld version expands the Structural implementation into a longer example where the names of classes reflect an actual scenario. Both these versions use very little in the way of new language features, sticking to inheritance and interfaces for expressing the relationships between classes — essentially representing object-orientation at the Java or C++ level. Where it is very difficult to implement a pattern at this level in a short example (for example, deep copy in the Prototype pattern), the functionality is left out. A NETOptimized solution is a rework of the RealWorld version, using as many C# 2.0 features as are fitting. Since the programs have not been updated since 2006, they do not include any features new to C# 3.0.

The implementations in Bishop [5] had the specific aim of exploring new language features. They also come in two versions, known as the Theory code and Example code. The Theory code is similar in length and intent to the Structural versions from DoFactory, and presents a minimalist version of each pattern, in which the essential elements can be seen in stark relief. The Examples add flesh to the pattern, and in many cases use more or slightly different features as a result.

Table 4 itemizes those pattern implementations in DoFactory's NETOptimized and Bishop's Example sets that use advanced C# features. The patterns are sorted according to features used, from the left. Those patterns that are omitted — Builder, Decorator, Factory Method, State, Strategy, Interpreter, Fa cade, Template Method, Flyweight — did not use any of the mentioned features in either implementation.

All of the pattern implementations (both those mentioned in Table 4 and the rest) make use of normal OOPS features such as inheritance, overriding, composition, access modifiers and namespaces, as well as other C# features such as properties, indexers, structs and (in the case of the Bishop set) object and array initializers. Consider now what Table 4 reveals.

1. Delegates, generics, iterators and nested classes are the language features that are exercised by 10 of the patterns.
2. The .NET features of Serializable and Reflection are used by three patterns.
3. The C# 3.0 features of extension methods and query expressions make an appearance in two patterns.

From this list, we can extract various pattern-feature pairs for investigation in terms of efficiency. In particular, those patterns that are already using a feature in one set of implementations, but not the other, are excellent candidates. For example, we can consider delegates in the Adapter pattern or query expressions (LINQ) in the Iterator pattern. The next section presents several implementations of the Visitor patern taken from these sources and others, and compares them for efficiency.

	delegate/event	generics	iterator	nested class	Serializable	Reflection	extension methods	query expressions
Adapter	y							
Command	y							
Mediator	y							
Chain	x	x						
Observer	x y	x						
Abstract Factory		y						
Composite		x y						
Iterator			x y					y
Proxy				x y				
Singleton				x y				
Memento					x y			
Prototype					x y			
Visitor						x		
Bridge							y	

Table 4. Advanced C# features in design patterns: x indicates DoFactory's NETOptimized implementations, y indicates Bishop's set

4 Experimenting with the Visitor pattern

The Visitor pattern defines and performs new operations on all the elements of an existing structure, without altering its classes. The pattern has two distinct parts: there are the classes that make up an object structure, and then there are the operations that will be applied to the objects in the structure. The complexity of the Visitor is enhanced by the fact that the object structure is usually assumed to have a variety of classes (often hierarchical) and different visit operations will be applicable for each type. There can also be different visitors, potentially traversing around the structure at the same time. Thus the language features required for its implementation revolve around type matching at runtime to find the correct `Visit` method.

There have been numerous studies of the Visitor pattern over the years, some of which are mentioned in Section 2.2 [17] [15]. What we present here is a fresh look at the pattern with four implementations, two of which make use of the features in Table 4 and two of which stick to ordinary OOP features

While our prime aim is to reveal the efficiency of implementations that make use of different language features, we need to balance these results against the other non-functional requirements mentioned in Section 2: readability, writeabil-

ity, maintainability and traceability. We introduce these, as we describe the four implementations.

4.1 Double dispatch

The classic technique for implementing the Visitor pattern follows three steps. Each data type is made visitor-ready by adding an identical `Accept` method:

```
public override void Accept(IVisitor visitor) {
   visitor.Visit(this);
}
```

Then there is an interface that lists a version of `Visit` for every possible data type. When a data type is added, the interface has to be amended.

```
interface IVisitor {
   void Visit (Element element);
   void Visit (ElementWithLink element);
}
```

The content of the visiting code is in methods all called `Visit`, defined for each data type in the structure. The content of the methods is part of the application, not the pattern.

Once that is all in place, the method that traverses the data structure can call `Accept` on objects of structure and dynamic binding will ensure that the correct `Accept` and the correct `Visit` method is invoked. This is the double dispatch mechanism. A simple program would be:

```
public void CountElements(Element element) {
   element.Accept(this);
   if (element.Link!=null) CountElements(element.Link);
   if (element.Next!=null) CountElements(element.Next);
}
```

The effort required to set up this scaffolding for double dispatch is quite daunting, and works against all the goals of traceability, readability, writability and maintainability [6]. The type matching is done at runtime, but directly through the virtual method tables kept for each object. Its advantage therefore is that there is very little hidden overhead.

4.2 Reflection

A completely different implementation is to let the runtime system search for the type, using the metadata available through reflection. The Visitor includes the following standard dictionary and method:

```
static Dictionary<Type, MethodInfo> methods =
             new Dictionary<Type, MethodInfo>();
```

```
public override void Visit(Element x) {
   Type type = x.GetType();
   if (!methods.ContainsKey(type)) {
      Type[] types = { type };
      methods[type] = this.GetType().GetMethod("Visit", types);
   }
   if (methods[type] != null)
      methods[type].Invoke(this, new object[] { x });
   else
      throw new Exception("no Visit method found");
}
```

This `Visit` method takes it upon itself to find out which of the actual `Visit` methods need to be called. It searches through the metadata available regarding the signatures of the methods called `Visit`. This version is an improvement over than in the Dofactory, in that it caches the method reference once found, and will not go through the look up process again. The cache is a generic Dictionary indexed by the type of x.

The reflection implementation is non-invasive to both the data structure and the Visitor. However, reflection is expensive and the approach suffers from a severe speed overhead.

The use of reflection to implement the Visitor pattern was previously investigated by Palsberg and Jay [18], among others. Their solution encapsulated the reflection inside a library class which they called the Walkabout. A recent advance over the Walkabout by Grothoff [13] is the Runabout class. It generates bytecode at run-time to make the execution much faster — only about 2 to 10 times slower than double-dispatch once the overhead of creating the bytecode has been incurred.

4.3 Type Testing

A variation on double-dispatch which avoids invading the data structure is type testing (also called an extrinsic visitor [17]). The choice of the correct `Visit` method is done by a sequence of if statements in the application itself. Using the same program as before, the application's method will be:

```
public void CountElements(Element element) {
   if (element is ElementWithLink)
      Visit(element as ElementWithLink); else
   if (element is Element)
      Visit(element as Element);
   if (element.Link!=null) CountElements(element.Link);
   if (element.Next!=null) CountElements(element.Next);
}
```

Here it is the application that is polluted with details pertaining to the data structure. If the number of data types is large, and if the data structure is to

be visited from more than one place in the application, this type testing can become tedious to write and maintain. A variation of this approach is to go back into the data types and have them maintain a class ID that can be used in a switch statement, somewhat faster than cascading if statements [17].

4.4 Delegates

Finally, we investigated the place of delegates in this pattern. Using a delegate, we don't have to pass the Visitor instance as a parameter into the data objects — the delegate contains a binding to both the Visitor instance and the method inside that instance. Then we replace a virtual dispatch `v.Visit` with a delegate call.

In each type in the data structure we add the following delegate and property:

```
public static VisitorDelegate vd;
public override VisitorDelegate VD { get {return vd;}}
```

Unfortunately, we cannot use the neat automatic property of C# 3.0 (where the get and set actions are generated by the compiler) because the property is overriding the property in the base type, and also giving access to a static field. The two unfortunately don't mix.

At the start of the Visitor, the delegates are set up, as in:

```
Element.vd = TallyFunction;
ElementWithLink.vd = delegate (Elements e) {
   Console.WriteLine("not counted");
};
```

where `TallyFunction` is a method of the `Visitor` class which visits an instance of type `Element`.

The advantage of the delegate method in terms of writeability is that the `Visit` methods can have different names, i.e. they do not all have to be called `Visit`, and also trivial visit functionality does not need a method at all: it can be expressed as an anonymous function. The above example shows both options. Thus, when using delegates, one can tailor one's code more to the real visiting, and not make it so stylised.

4.5 Results

We evaluate the Visitor implementations in two ways: for efficiency and for the non-functional properties.

In our experiments, the data structure being visited is a tree where there are N choices of the data type for a non-leaf node and L choices for the data type of a leaf node. All non-leaf nodes have two children. The tree was generated as a balanced tree of depth 10 in every case (so the tree contains 1023 nodes in total), where the data types of the nodes were selected randomly from the possible choices.

All times in Table 5 are reported as ticks per visited node, where one tick is equal to 100 nanoseconds. The times are measured over 100 traversals of the tree on an Intel Core 2 Duo running at 2.67GHz. Each visit method performs minimal work — just incrementing a counter. If visit methods did some real work (or performed I/O) then the timing differences would be obscured. In order to eliminate JIT effects from the timings, the test program performs one traversal of the data structure before starting the clock. All invoked methods are therefore JIT'ed in advance.

Tree Classes			**Visitor Implementation**			
N	**L**	**N+L**	**DD**	**TT**	**DE**	**RC**
1	1	2	10	19	11	3791
2	1	3	10	23	12	3782
2	2	4	10	24	11	3809
3	2	5	11	30	13	3748
4	2	6	12	36	13	3852
4	3	7	14	39	16	3832
5	3	8	15	43	16	3784

N = Number of non-leaf classes
L = Number of leaf classes
DD = Double-dispatch implementation of visitor
TT = Sequence of type-tests to select visitor method
DE = Delegates to select visitor method
RC = Reflection with a cache to select visitor method

Times measured as ticks per node; 1 tick = 100 ns

Table 5. Visitor pattern timings

Both the double-dispatch and the delegate approach perform well. As long as the number of classes in the data structure is reasonable (say in the tens or twenties), the sequence of type-tests approach is probably good enough. (It should be noted that the `is` test is not necessarily fast.) However the implementation of reflection is comparatively so expensive that it should probably not be advocated either for large structures or for time critical applications.

The non-functional properties are a combination of the factors listed in Table 6.

In terms of non-functional properties, the reflection cached approach is a clear winner as it can add a visitor framework onto existing code. However, if building from scratch, the delegate version has much to recommend it, in that it is sensitive to the real code of the Visitor, in terms of naming and structure. The double-dispatch version, on the other hand, imposes a regimen of `Accept` and `Visit` methods, including the names.

	Double Dispatch	**Type Test**	**Delegates**	**Reflection Cached**
Invasive of the data structure	Yes — add a standard method	No	Yes — add a delegate and property	No
Invasive of the Visitor	Interface listing a `Visit` method for each type	Only the type tests, embedded in the Visitor code	Delegate initialization in the constructor	Single standard `Visit` method containing a reflective test
Constraining the visitor	A `Visit` method must exist for each type	No — visit methods can have any name and be omitted	No — Visit methods can have any name and be omitted	All methods must be called `Visit`, but can be omitted
Main Disadvantage	Unnecessary empty `Visit` methods	Type tests might need to be repeated in the Visitor code	Cannot easily have two Visitors running together	Significant hidden overhead due to use of reflection
Main Advantage	No hidden overhead	Integrated with Visitor code	Works with the Visitor code to be efficient	Almost transparent to Visitor and data structure

Table 6. Visitor method implementation comparison

We can also extend the Delegate implementation to handle multiple kinds of Visitor. Add a static list of delegates in each data structure class. Then each Visitor, when constructed, takes the next slot in each visitee class's array to fill in the delegate reference and remembers the index of that slot. The VD property in each data structure class becomes:

```
public static List <VisitorDelegate> vd;
public override VisitorDelegate VD (int slot)
   {get {return vd[slot];}}
```

There will be a small performance hit with the generalized approach because of the cost of passing the extra parameter and of indexing the delegates list. However, the changes are standard throughout the methods.

5 Conclusions and Future Work

This is the first time that a comparative study of patterns aimed at non-functional attributes has been made, and it brings up to date the work that was started more a decade ago. We have surveyed the various approaches to identifying the patterns that are amenable to new language features, and then tested how this is progressing in 2008 by using two sets of available pattern implementations. Homing in on one of the complex patterns, the Visitor, we conclude that new features such as delegates, generics and properties, when used together, can make for a readable implementation which has an acceptable efficiency overhead (the DE or delegates implementation).

In work ongoing we are examining the other patterns in the same way, and producing a bank of results. One problem we face is in isolating a testbed for a pattern, since some of them simply do not work well unless there is a real world harness in place. We shall therefore look at how patterns are used and position our experiments both in an image of the real world, and in an abstract environment.

Acknowledgements. Our thanks to Pierre-Henri Kuaté who ran the programs to obtain the results, and to him, Rhodes Brown and Stefan Gruner for helpful discussions. This work was supported by grants from the National Research Foundation of South Africa and the Natural Sciences and Engineering Research Council of Canada.

References

1. Agerbo, E., and Cornils, A.: How to Preserve the Benefits of Design Patterns. Proc. OOPLSA, pp. 134–143 (1998)
2. Arnout, K., and Meyer, B.: Pattern Componentization: the Factory Example, Innovations in Systems and Software Technology: A NASA Journal 2, (2), pp. 65–79, (2006).
3. Baumgartner, G., Läufer, K., and Russo, V.F.: On the interaction of object-oriented design patterns and programming languages. Technical Report CSR-TR-96-020, Purdue University (1996).
4. Bishop, J.: Language features meet design patterns: raising the abstraction bar. Workshop on the Role of Abstraction in Software Engineering (ROA 08), co-located with ICSE 2008: to appear.
5. Bishop, J.: C# 3.0 Design Patterns. O'Reilly Media, Sebastapol, CA, 2008
6. Bosch, J.: Design Patterns as Language Constructs. Journal of Object-Oriented Programming 11, 2, pp. 18–32 (1998)
7. Chambers, C., Harrison, W., and Vlissides, J.: A Debate on Language and Tool Support for Design Patterns. Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 277–289 (2000)
8. Data and Object Factory, Design Pattern Framework: C# Edition. `http://www.dofactory.com/Default.aspx` (2006)
9. Frick, A., Zimmer, W., and Zimmermann, W.: On the Design of Reliable Libraries. Proc. of TOOLS 17, pp. 13–23 (1995)

10. Friedrich, M., Papajewski, H., Schröder-Preikschat, W., Spinczyk, O., Spinczyk, U.: Efficient Object-Oriented Software with Design Patterns. Proc. of Symposium on Generative and Component-based Software Engineering, (GCSE 99), LNCS 1799, pp. 79–90 (2000).
11. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA, Addison-Wesley (1995).
12. Gil, J., and Lorenz, D.: Design Patterns *vs* Language Design. Proc. of Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), pp. 108–111 (1997)
13. Grothoff, A.: The Runabout. Software: Practice and Experience, *t*o appear (2008)
14. Hannemann, J., and Kiczales, G.: Design Pattern Implementation in Java and AspectJ. Proc. of OOPSLA, pp. 161–173 (2002)
15. Meyer, B., and Arnout, K.: Componentization: the Visitor Example. Computer 39, (7), pp. 23–30 (2006)
16. Microsoft Corporation: C# 3.0 Reference Documentation, `http://msdn2.microsoft.com/vcsharp`
17. Nordberg III, M.E.: Variations on the Visitor Pattern. Proc. of Workshop on Pattern Languages of Programming (PLoP), (1996).
18. Palsberg, J., and Jay, C.B.: The essence of the Visitor Pattern. Proc. 22nd IEEE Int. Computer Software and Applications Conf. (COMPSAC), pp. 9–15, (1998)