Journal of Universal Computer Science, vol. 12, no. 7 (2006), 849-867
submitted: 28/1/06, accepted: 6/4/06, appeared: 28/7/06 © *J.UCS*

The Language of the Visitor Design Pattern

Markus Schordan
Institute of Computer Languages
Vienna University of Technology, Austria
markus@complang.tuwien.ac.at

Abstract: Design patterns have been developed to cope with the vast space of possible different designs within object-oriented systems. One of those classic patterns is the Visitor Pattern, used for representing an operation to be performed on the elements of an object structure. In general, the order in which the objects are visited is crucial. We present a mapping from the Visitor Pattern to a context free grammar that defines the set of all such visit sequences, a given Visitor can perform. The language defined by this grammar is the language of the Visitor Design Pattern and the mapping encodes knowledge about the class hierarchy and the implementation of the accept methods of a Visitor Design Pattern. It is general enough to model complications that occur when traversing arbitrary object structures, and also properly represents cases such as lack of a common base class, multiple inheritance, and inheritance from concrete classes. Due to its particular design, the grammar can also be used as precise documentation for a Visitor Design Pattern.

Key Words: context free grammar, visitor pattern, visitor language, method invocation sequence, essential aspect
Category: D.1.5, D.3.1, F.4.2

1 Introduction

Design patterns have been developed to cope with the vast space of possible different designs within object-oriented systems. Naming the patterns created a terminology that is used for describing such systems. From our work on optimizing the use of high-level abstractions in applications it has become apparent that ruminations on programming languages and libraries often lead to the well-known insight, to which a whole chapter is devoted in [1] "Library design is Language Design" and "Language design is library design".
When we are faced with a library, an abstraction in a library, or in particular a design pattern - rarely, if ever, we can specify a "language" that would materialize as an indicator of a deepening understanding of the design of the abstraction. Such a language should represent the *essential aspect* of the pattern. If we can define such a language, we can claim that "pattern design is language design". Consequently, this would mean that whenever we create and use a pattern, we implicitly also create a language. And it could also contribute to the discussion, whether it is worthwhile to invent new languages. If we can understand pattern design as language design, it becomes permissible to say that pattern designers

are also language designers. And then it is eligible to say that many such languages have been developed in recent years; and have been accepted and did survive in form of patterns.

Our contribution is the definition of such a language for one of the classic design patterns presented in [2]. The pattern of interest is the Visitor Design Pattern. The *essential aspect* of this pattern is that it defines a traversal on an object structure and "visits" each node of that structure in some defined order. Such a structure can be potentially infinite, and the visited objects do not have to have a common base class.

The declared purpose of our contribution is the precise documentation of a Visitor. Whereas it is commonplace to generate visitor code from AST grammars, we formalize a way to extract a grammar from code in which the Visitor Pattern can be identified. The proposed mapping from a Visitor to a grammar is a contribution to the field of documenting Visitors, and opens the possibility of applying the technique to other types of Design Patterns. To allow the application of the presented approach to data structures in general, the formalism handles complications that occur when traversing arbitrary object structures, such as lack of a common base class, multiple inheritance, and inheritance from concrete classes. The mapping encodes knowledge about the class hierarchy and the implementation of the accept methods of a Visitor.

We present how to map the invocation of the visit methods of a Visitor Pattern into a context-free grammar that generates all the sequences of visit method invocations, thus providing a high-level view of all possible visiting sequences that can be performed by the Visitor. We shall call the language generated by that grammar the language of the Visitor Design Pattern, or short Visitor Language. A word of the language corresponds to the sequence in which the nodes of the object structure are visited, i.e. a terminal represents the invocation of a visit method; and the set of all those words, the language, represents all possible visit sequences that may be performed given any object structure for which the Visitor is implemented to perform on.

The definition of the Visitor Language is an attempt to building a bridge between grammar-oriented approaches and software design. There exists a broad range of contributions to the field of grammar-based specification of object-structures and automatic generation of Visitors [3–12]. A direct correspondence of grammars and the special case of object-oriented abstract syntax trees has also been discussed in [13] by Appel. Whereas these approaches are based on a grammar and generate the interfaces and implementation of the object structure and a Visitor, we define the language for an existing Visitor Pattern, literally going into the opposite direction. This is motivated by our work on abstract aware analysis for automatic recognition of abstractions and generation of documentation and annotations. Generating a grammar permits arguing that we also find "hidden"

Listing 1: **Abstract C++ class Visitor and inheriting class MyVisitor.**

```
class Visitor {
public:
  virtual void visitB(B*)=0;
  virtual void visitC(C*)=0;
  virtual void visitD(D*)=0;
  virtual void visitE(E*)=0;
};
```

```
class MyVisitor : public Visitor {
public:
  void visitB(B* obj) { cout << "b" << endl; }
  void visitC(C* obj) { cout << "c" << endl; }
  void visitD(D* obj) { cout << "d" << endl; }
  void visitE(E* obj) { cout << "e" << endl; }
};
```

languages in software. In particular, it is the recent attempt of going towards the discipline Grammarware [14] that fuels our endeavour of generating a grammar for existing patterns such that patterns can be understood as languages. It also permits making the argument in the other direction, that is, what could have been saved in development if grammars would have been used to specify and generate the code. At least, it documents a Visitor Design Pattern such that the traversal it can perform, is precisely documented for users.

Since the understanding and appreciation of a language is fundamentally connected to its design and how "easy" it is for users to learn, apply, and use, we also consider the design of our grammar as an important ingredient for its appeal. The design of the grammar is as critical as the design of the software pattern that we are mapping from. We are going to define a grammar that is as appealing and easy to read as the original pattern. Actually, we attempt to go beyond that. It should be easier to read, permitting to focus on the *essential aspect* of the Visitor Pattern only, formalizing only the relevant information that constitutes a Visitor Pattern. But we do not present a new formalism, or a new extended form of a grammar. The grammar we use is a context free grammar, with only two specific forms of productions such that each form of production represents distinct properties of the Visitor Pattern.

In the following section we present our running example that will serve in explaining several properties and details of the mapping in later sections as well. In the example we also present the corresponding formal grammar as defined by our mapping. The details on how this grammar is obtained in general is discussed in subsequent sections.

1.1 Example

The Visitor Pattern is used for representing an operation to be performed on the elements of an object structure. A Visitor essentially does a depth-first-walk of an object structure, executing an "action" method, usually called `visit`, at each object. It is a frequently used pattern in libraries that implement object structures.

Listing 2: C++ class interfaces of traversed object structure.

```
class A {
public:
  virtual void accept(Visitor& v)=0;
};

class B : public A {
public:
  B(A* next0,D* data0)
    :next(next0),data(data0) {}
  virtual void accept(Visitor& v);
private:
  A* next;
  D* data;
};

class C : public A {
public:
  C(){}
  void virtual accept(Visitor& v);
};

class D {
public:
  D(){}
  virtual void accept(Visitor& v);
};

class E : public D {
public:
  E():D(){}
  virtual void accept(Visitor& v);
};
```

Listing 3: Implementation of accept methods of object structure.

```
void B::accept(Visitor& v) {
  v.visitB(this);
  next->accept(v);
  data->accept(v);
}

void C::accept(Visitor& v) {
  v.visitC(this);
}

void D::accept(Visitor& v) {
  v.visitD(this);
}

void E::accept(Visitor& v) {
  v.visitE(this);
}
```

The example code in Listing 2 shows the interfaces of our example classes. We use a minimal artificial example that is designed to ease the demonstration of formal properties of the mapping. Although small, the example includes two properties that require particular attention. One property is that the class hierarchy that is used for the object structure, does not have a common base class from which all other classes inherit. And the second one being that we also have a class inheriting from a concrete class. The latter second case for example, is not present in the approach taken by Appel in [13] for a grammar based definition of an abstract syntax tree (AST). For ASTs Appel's design results in a good design of the tree, where concrete classes only exist as leaf nodes in the class hierarchy. But since we present a mapping for the general case, and do not want to suggest

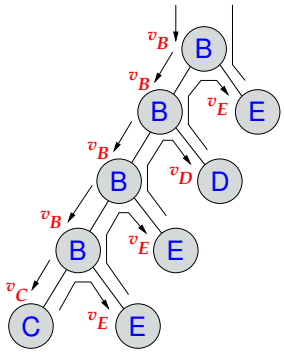

Figure 1: Visualization of a concrete object structure and its traversal with the given example Visitor. Each arrow head corresponds to a visit of the respective node, denoted v_X, where X is the type of the object. The visit method invocation sequence for this concrete object structure is $v_B^4 v_C^1 v_E^2 v_D^1 v_E^1 \in L(G_1)$.

to change the design in any way, we need to incorporate such cases as well.
We have one abstract base class, `A`, and two concrete classes, `B` and `C` inheriting from class `A`. And we have another second class hierarchy, with class `D` as root class of the hierarchy, and one class `E` inheriting from `D`. Both classes, `D` and `E`, are concrete classes.
If we create a data structure with the objects of type A to E, we can use a Visitor to traverse that data structure and specify a visit method for each concrete object, see Listing 1. The other part of the Visitor Pattern is the accept methods shown in Listing 2 and Listing 3. We shall show that we can create a formal grammar that defines the set of all sequences of visit methods that can be invoked by a Visitor. The formal grammar for the example Visitor in Listing 2 and Listing 3 is

$$G_1 = (N_1, T_1, P_1, A) \text{ with } N_1 = \{A, B, C, D, E\}, T_1 = \{v_B, v_C, v_D, v_E\},$$
$$P_1 = \{A \to B, A \to C, B \to v_B\ A\ D, C \to v_C, D \to v_D, D \to E, E \to v_E\}$$

where N_1 is the set of nonterminals, representing the abstract and concrete classes used for defining the object structure. The set of terminals is T_1 where we use the notation that v_X represents a call of the visit method for class X.
Grammar G_1 generates the language

$$L(G_1) = \{x^n y z^n \mid x = v_B, y = v_C, z \in \{v_D, v_E\}, n \geq 1\},$$

a deterministic context free language. This is the set of all sequences of visit

Listing 4: **Alternative implementation of the accept method of class B.**

```
void B::accept(Visitor& v) {
  v.visitB(this);
  data->accept(v);  /* declared type of data is D* */
  next->accept(v);  /* declared type of next is A* */
}
```

methods that can be called, if the example Visitor is used by invoking the accept method of an object referred by a variable with declared type A. In the above example, the visit methods are implemented, see Listing 1, such that MyVisitor prints a lower caps letter of the class to which the visit method corresponds to. Hence, the visit method for class B, `visitB`, prints 'b' to stdout. We use that to illustrate our pattern mapping by example. The sequence of invoked visit methods is reflected in the output of the example code as well, and the set of all possible outputs of our Visitor is $\{x^n y z^n \mid x = b, y = c, z \in \{d, e\}, n \geq 1\}$.

Please note that $n \geq 1$. For our language definition we assume that traversing the object structure does not cause errors such as dereferencing of null pointers or exceptions. Here this means that at least a concrete object of type `B`, one object of type `D` or `E`, and one of type `C` must exist. In general, we assume that the Visitor succeeds traversing a given object structure. Otherwise we would need to consider the set of all prefixes of all words which would render the language rather useless. We shall discuss that aspect, sub languages, and variations of the classic pattern in Section 2.5.

One traversal of a concrete object structure corresponds to one word in the Visitor Language. In Fig. 1 a concrete object structure with nine objects is shown. Starting at the root node, the sequence of visit method invocations for this concrete structure and the given Visitor is $v_B^4 v_C^1 v_E^2 v_D^1 v_E^1$. This sequence is a word of the Visitor Language $L(G_1)$. The set of all visit method invocation sequences that can be performed by the given Visitor is the language defined by grammar G.

Now let us also determine the Visitor Language with an alternative implementation of one of the accept methods of this Visitor. We use this variation to demonstrate that the Visitor Language is different, dependent on the implementation of accept methods. Let us reverse the two invocations of the accept methods in class B's accept method as shown in Listing 4. Mapping this alternative Visitor to a grammar gives us a new grammar, G_2, where only the set of productions is different to G.

$$G_2 = (N_1, T_1, P_2, A), \text{ with } P_2 = P_1 - \{B \to v_B\ A\ D\} + \{B \to v_B\ D\ A\}$$

The difference, representing the fact that the accept method of class B is implemented differently, is that we have now the production $B \rightarrow v_B\ D\ A$ instead of $B \rightarrow v_B\ A\ D$. Consequently, the language is also different,

$$L(G_2) = \{(xy)^n z \mid x = v_B, y \in \{v_D, v_E\}, z = v_C, n \geq 1\},$$

a regular language. We can write it as regular expression $(v_B(v_D \mid v_E))^+ v_C$, where '+' denotes that there exists at least one occurrence of the regular expression in brackets, and '|' that we have either v_D or v_E as terminal at this position. Thus, the language classes of our Visitor Language can be different, ranging from regular languages to context free languages.
In Section 2 we present a formal definition for the mapping from a Visitor Pattern to a grammar. In Section 3 we discuss various applications of our approach, followed by a comparison with the related work in Section 4. Eventually we conclude in Section 5 that our mapping permits understanding design pattern design as language design.

2 Mapping the Visitor Pattern to a Formal Grammar

In this section we shall define a mapping between the Visitor Design Pattern to a formal grammar such that the grammar generates the set of all visit sequences that the visitor can successfully perform. With "successfully" we mean that the Visitor does not fail because of errors in the object structure.
The mapping consists of two parts. First, we need to determine the relevant information of the implementations of all accept methods of a Visitor Pattern. Second, we shall determine the relevant information of the class hierarchy.

2.1 Accept Methods

The relevant information of the implementation of an accept method of a class, A, is

i The name of the class, A,

ii The name of the visit method that is invoked for class A, denoted v_A,

iii The order in which other accept methods are invoked by the accept method of class A. We shall map the declared types of the variables on which the accept methods are invoked to a sequence of grammar symbols, denoted $A_1 \ldots A_n$.

Let the set of classes with accept methods of the (same) classic Visitor Pattern be $\mathcal{C}$. We define the set of productions, P_a, mapped from the accept methods as

$$P_a = \{A_0 \rightarrow v_A\ A_1 \ldots A_n \mid A_i \in \mathcal{C}, \texttt{is\_concrete}(A_0), n \geq 0, 0 \leq i \leq n\}$$

On the left-hand-side of a production we have a nonterminal representing the name of the class, A_0. For P_a we only consider concrete classes for the left-hand-side with the predicate `is_concrete` holding only for classes with an implementation of an accept method. The right-hand-side has as first grammar symbol a terminal, v_A, which represents the invoked visit method for class A; the visit method is usually called `visit` followed by a class name. This terminal is followed by a possibly empty list of nonterminals $A_1 \ldots A_n$, each representing the declared type of the variable on which the accept method is invoked. If n is not fixed, we denote this using the notation from regular right part grammars, as A_i^*. This is the case for containers of objects which can be of arbitrary size. It is used for the Composite Pattern.

For example, let us apply this mapping to the accept method of class `B` in Listing 3. As relevant information we obtain `B` (i), $v_{\texttt{B}}$ (ii), and the list `A,D` (iii). Hence, we define the production $B \rightarrow v_B\ A\ D$. For the other three classes `C`, `D`, `E` we obtain $C \rightarrow v_C$, $D \rightarrow v_D$, $E \rightarrow v_E$.

Thus, the left-hand-side corresponds to the name of the class with the accept method in question. The right-hand-side corresponds to the implementation of the accept method, i.e., in which order the visit method and the accept methods of other classes are invoked. The order of the other classes is represented by the order of the grammar symbols that correspond to the declared types of the variables on which the accept methods are invoked.

We can now define a grammar, G_a, that represents the language generated by the above mapping of accept methods. Let N_a be the set of all nonterminals existing in P_a either on the left-hand-side or right-hand-side and let T_a be the set of all terminals v on the right-hand-side in any production in P_a. Let the start symbol of our grammar be a nonterminal corresponding to a class with an accept method, S_a.

$$G_a = (N_a, T_a, P_a, S_a)$$

An interesting property of grammar G_a is that it is in Greibach Normal Form [15]; this form is often used as basis for formal proofs on grammars. Here it defines the set of all visit-sequences that can be generated by calling the accept method of an object in an object structure. In the next section we extend this grammar with productions representing the case that the Visitor is invoked on a variable with declared type of some abstract base class.

2.2 Class Hierarchy

The classes of an object structure do not necessarily have to have a common base class. This is case is represented in our running example in Listing 2 with class `D` not being in the class hierarchy of classes `A`, `B`, and `C`.
It remains to define the productions mapped from the class hierarchy, to complete the grammar of the Visitor Language. Let the predicate `is_base_class`(A, B) hold, if class A is a (direct) base class of class B and class A has an accept method declared. We define the set of chain productions, P_c, for classes with an accept method of the (same) Visitor as

$$P_c = \{A \to B \mid \texttt{is\_base\_class}(A, B), A \in \mathcal{C}, B \in \mathcal{C}\}$$

For example, for our Visitor Pattern in Listing 2 we obtain as corresponding set of chain productions $\{A \to B, A \to C, D \to E\}$. These productions correspond to the inheritance in the class hierarchy where the accept method of the Visitor Pattern in question is inherited.

2.3 Complete Grammar

The complete Grammar, G, is composed from the defined mapping of the accept methods and the classes with an accept method in the class hierarchy. Let N_c be the set of all nonterminals on any side of the productions in P_c. Note that P_c has no productions with terminals. Then we define the complete grammar as

$$G = (N_a \cup N_c, T_a, P_a \cup P_c, S_c)$$

The set of nonterminals, $N = N_a \cup N_c$, consists of the union of nonterminals of the productions generated from the relevant information in the accept methods and classes with accept methods of the class hierarchy. The terminals, T_a, are those defined in the mapping of the accept methods only. The productions are, similar as the nonterminals, the union of P_a and P_c. The start symbol can be any nonterminal corresponding to a class with an accept method to generate the set of visit sequences of the accept method of that class. The set of productions as defined above, allows to derive any possible visit sequence by choosing the appropriate start symbol.
The grammar therefore only has two kinds of productions of the form

1. $N_0 \to N_1$ (corresponds to inheritance)
2. $N_0 \to v_N \; N_1 \ldots N_n$ with $n \geq 0$. (corresponds to accept methods)

which permits defining the language of a Visitor Design Pattern.

Relevant information of running example	Mapped Productions
B::accept(...) {	
visitB(...);	
next->accept(...); :next is of declared type A∗	
data->accept(...);...} :data is of declared type D∗	$B \rightarrow v_B\ A\ D$
C::accept(...){ visitC(...) ... }	$C \rightarrow v_C$
D::accept(...){ visitD(...) ... }	$D \rightarrow v_D$
E::accept(...){ visitE(...) ... }	$E \rightarrow v_E$
class A { accept(...); ...}	
class B : public A { accept(...); ...}	$A \rightarrow B$
class C : public A { accept(...); ...}	$A \rightarrow C$
class D { accept(...); ...}	
class E : public D { accept(...); ...}	$D \rightarrow E$

Figure 2: Table showing how the presented mapping is applied to the running example. Only that information of the source code is shown that is relevant for mapping the Visitor to grammar G_1. The first block of rows in column two shows the productions for P_a of P_1, the second block of rows the productions for P_c of P_1.

Eventually we show in Fig. 2 for our running example from Listing 2 and Listing 3 how we obtain the grammar presented in the introduction with the above definition. In the first column only that portion of the source code is shown that is relevant for the mapping. It is the accept methods, and the inheritance hierarchy for those classes. Note that class `D` is a base class but also a concrete class in the object structure and therefore both kinds of productions exist with D on the left-hand-side.

2.4 Readability and Direct Correspondence

The readability and direct correspondence to the implementation of the Visitor Pattern is important such that the grammar can be used for documenting an existing Visitor. The automatic grammar generation for an existing Visitor we shall discuss in the next section. Here we shall highlight the expressiveness of our defined formal grammar.
The grammar has only two kinds of productions, chain productions and productions starting with a terminal. Each kind represents exactly one property of the Visitor Pattern.

2.4.1 Chain Productions.

A chain production is of the form $A \rightarrow B$ where A is a base class of B. It represents inheritance (with the arrow going into the opposite direction as in the UML notation for class hierarchies). This production exists in the grammar because there exists an accept method in class A and it is inherited by B. If multiple classes inherit from A we have multiple chain productions.
In particular, *all* is-a relationships that are relevant to the Visitor Pattern are represented by such chain rules. Note that this permits representing multiple inheritance as well.

2.4.2 Productions with a Terminal.

The second kind of productions is of the form $A \rightarrow v_A\ A_1 \dots A_n$. The number of this kind of productions is exactly the number of concrete classes that are relevant to the Visitor Pattern. Such a production never represents inheritance; it can be understood as a has-a relationship if we wish to have the object structure design in mind and the accept methods always directly reflect that class A has members of type $A_1 \dots A_n$.
From such a production we know that there exists a concrete class A, that the visit method v_A is invoked in the accept method of class A, and that the traversal proceeds by traversing objects of type A_1 to A_n, in the specified order. The order in which the visit and accept methods are called is directly reflected in the order of the grammar symbols on the right-hand-side of the production.

2.4.3 Language and Start Symbol of the Grammar.

Because the terminal v_X directly corresponds to an invocation of a visit method, the language generated by the grammar is the set of all traversals, or in other words, the set of all sequences of invocations of the visit methods. A word of the language directly corresponds to a traversal.
The start symbol of the grammar corresponds to the declared type of the variable that holds the reference to the first object being traversed. Any nonterminal of the grammar can be chosen as start symbol because every nonterminal on the left-hand-side directly corresponds to a class of that name with an accept method.

2.5 Variations of the Classic Pattern

Variations of the Visitor Pattern can be represented as well. In a post-order traversal, v_X is the last grammar element on the right-hand-side. For an extension of the Visitor Pattern, such as performing a `preVisitX` and a `postVisitX`, we have two (distinct) terminals on the right-hand-side, v_X and v'_X.

Listing 5: **Alternative implementation of a variation of the Visitor Pattern with preVisit and postVisit methods in class B's accept method.**

```
void B::accept(Visitor& v) {
  v.preVisitB(this);      /* first visit of node with type B */
  next->accept(v);        /* declared type of next is A* */
  data->accept(v);        /* declared type of data is D* */
  v.postVisitB(this);     /* second visit of node with type B */
}
```

Listing 6: **Alternative implementation of a variation of the Visitor Pattern with a null pointer check in class B's accept method.**

```
void B::accept(Visitor& v) {
  v.visitB(this);
  next->accept(v);
  if ( data ≠0 ) {      /* check whether data points to an object */
    data->accept(v);
  }
}
```

The corresponding production for the alternative implementation in Listing 5 is

$$B \rightarrow v_B \ A \ D \ v'_B$$

We shall discuss a concrete example of this variation from our own work in Section 3.1. This variation of the Visitor Pattern is also utilized in [8] for computation of inherited attributes (pre-visit) and synthesized attributes (post-visit).
Another variation of the classic Visitor Pattern is the use of null pointers and having accept methods check whether a pointer is null. This fact can be represented by making the corresponding grammar symbol optional on the right-hand-side of a production. In Listing 6 such an alternative implementation is shown for our running example's accept method of class B. Written as regular right part grammar, the corresponding production to this alternative implementation is

$$B \rightarrow v_B \ A \ (D \mid \epsilon)$$

The Visitor Language for our running example in section 1.1 with this alternative implementation of class B's accept method, is the language $v_B^n v_C^1 (v_D \mid v_E)^m$, where $n \geq m, n \geq 1, m \geq 0$.
If conditions are used to decide the order of the traversal, the language might actually be context sensitive. For that case we suggest extending the grammar to an attribute grammar for specifying the additional constraints in semantic actions but this requires further investigation in future work.

3 Applications

In this section we show in which fields the presented approach, using a mapping from a Visitor to a grammar, has already been applied in our own work. We also wish to make clear that the mapping can be fully automated by using existing source-infrastructures such as ROSE [16].

3.1 Grammar as Documentation of the Visitor Pattern

In ROSE we provide beside the classic Visitor also some variations of the Visitor Pattern that have proven suitable for advanced computations on the AST. We give a short example of the textual representation of the grammar that we use in ROSE for documenting the Pre-Post AST Visitor, a Visitor that visits a node twice, in a pre order traversal and a post order traversal (this variation of the classic pattern is also called before/after Visitor). The grammar is generated following the mapping presented in Section 2 and has been added to the ROSE reference manual. The entire class hierarchy of the ROSE C++ AST consists of 246 classes. The above mentioned Visitor is designed to visit only a subset of these, 171 in total. The information stored in non-visited nodes of the AST is available via access functions, which can be considered as accessing pre-defined attributes (such as type information, modifiers, etc.). Therefore a comprehensive and precise documentation of the Visitor is necessary – and the presented grammar has proven useful for that purpose. The ROSE AST has one common base class and uses inheritance from concrete classes.
In Fig. 3 we show a grammar fragment, generated as documentation for the ROSE Pre-Post AST Visitor. The terminals of the grammar are the names of the visit methods, for each node there are two visit methods, the preVisit and postVisit method. The prefix "Sg" of class names is used for historical reasons, because the ROSE AST is based on the Sage++ AST and "Sg" is an abbreviation for Sage. Note that we use the Kleene star '`*`' for specifying an arbitrary number of SgStatement node visits after pre-visiting a SgBlock node. Here the Kleene star actually represents an (internal) iteration on a C++STL container.
For example, if a user wants to know what sequence of visit methods the Visitor can perform when called on an AST object of type `SgScopeStatement`, he can see that `SgScopeStatement` is a virtual (abstract) class with a declared pure virtual accept method. Other classes, `SgBlock`, `SgIfStatement`, etc. inherit because chain productions exist with nonterminal `SgScopeStatement` on the left-hand-side. A `SgBlock` node is a concrete node because we have terminals representing preVisit and postVisit on the right-hand-side of the production with `SgBlock` on the left-hand-side. In case of inheritance from concrete classes, both kinds of productions exist with the same nonterminal on the left-hand-side.

```
SgStatement        :  SgScopeStatement
                   |  SgDeclarationStatement
                   |  ...
                   ;
SgScopeStatement   :  SgBlock
                   |  SgIfStatement
                   |  SgForStatement
                   |  ...
                   ;
SgBlock            :  preVisitSgBlock
                      SgStatement *
                      postVisitSgBlock
                   ;
SgIfStatement      :  preVisitSgIfStatement
                      SgStatement SgBlock SgBlock
                      postVisitSgIfStatement
                   ;
...
```

Figure 3: Grammar fragment example from the generated Visitor documentation in ROSE for the Pre-Post Visitor.

3.2 Grammar-Based Interoperability of Tools

We briefly describe an application going towards the Grammarware discipline as described in [14], based on our presented Visitor grammar. The Program Analysis Generator (PAG) [17] requires an abstract grammar as input, so called syn files. The abstract grammar specifies the Abstract Syntax Trees on which the generated program analyzer operates on.
When we integrated PAG into the C++ source-to-source infrastructure ROSE [16], we first generated the documentation for the AST Visitor. Here the classic Visitor was of interest. It travarsed the same subset of AST nodes as the above mentioned Pre-Post Visitor. This grammar was also input to another tool, called GRATO, to transform the grammar, by also pruning all control-flow related symbols, into another grammar, representing the abstract grammar (without control-flow relevant information) as required by PAG. Hence, the documentation of the Visitor also served as input format for generating an adapted grammar, as required by another tool, PAG.
PAG is used to specify program analyses based on abstract interpretation, which we use with ROSE for analyzing source code and detecting more advanced vari-

ations of Visitors in existing source code. Therefore, we can automate the detection of the Visitor Pattern and generate a grammar as presented, defining the Visitor Language. The details of the abstraction aware analysis are beyond the scope of this paper.

4 Related Work

The visitor pattern has been intensively studied, mostly from the perspective of specifying a traversal on an object structure and generating the implementation of the object structure and a Visitor for performing the traversal. All those approaches incorporate the use of a grammar at some point. The documentation generation for Design Patterns has also been addressed in [11, 18]
Recently, the most general approach has been defined by Klint et al. in [14]. They propose to incorporate the use of grammars at all levels in development and comprise grammar and all grammar-dependent software in the so called discipline Grammarware. In this discipline our presented grammar could be understood as a base-line grammar within the grammar life-cycle. This is, also from our perspective, the ideal case, to start with a grammar, and by transforming and extending that grammar we can generate other components of the software system. If the software already exists, and our approach is motivated by that setting, we need to create a grammar from existing source code. The same authors have also contributed in the field of semi-automatic grammar recovery [19] but focus on existing parsers and generating a concrete grammar.
A very general approach to the specification of traversals is presented by Lieberherr et al. in [5]. This approach supports structure-shy specification of traversals. Only those aspects are specified as constraints that are considered relevant to the traversal and the generator ensures that those constraints are met. This approach supports changes to the object structure, which is a general problem of the Visitor Pattern. Our approach does not attempt to contribute to the problem of changes to the object structure, our contribution is in the field of generating documentation for Visitors and presenting a grammar that can be automatically obtained from an existing Visitor implementation with our C++ infrastructure ROSE [16]. An approach for specifying recursive traversals is presented in [6]. It is based on traversal specifications that allow specifying traversals that can revisit the same node and also to dynamically control the behaviour of the traversal. In particular, it also permits calling other traversals within a traversal. This permits combining different traversals and abstractions of those. Visser has also contributed in this field by proposing Visitor combination for similar reasons in [4]. This work has been developed into a full framework, the JJTraveler [7], together with Arie van Deursen. In our approach we can express that by combining different grammars into one grammar. For example, instead of considering only

one Visitor we can consider a set of Visitors and all their accept methods. This gives us a single language for a set of combined Visitors.

An object-oriented view on attribute grammars that is similar to our grammar was already presented by Koskimies [20] in 1991. He used two notions of nonterminals, so called superclass nonterminals and basic nonterminals. The concept of superclass nonterminals and the use of chain productions to express the inheritance relation is the same as in our approach. But we do not use the concept of basic nonterminals to specify the syntactic composition of basic language constructs. In contrast, in our grammar only the invocation of a visit method corresponds to a terminal. A basic nonterminal on the left-hand-side of a production and the so called slots in [20] correspond in our grammar to productions corresponding to implementations of accept methods. A similar approach was also discussed by Grosch in [21], where he shows how with object-oriented attribute grammars common parts of a specification can be "factored out". Some tools take the approach, such as Alexey Demakov's TreeDL and Etienne Gagnon's SableCC, of using Visitors for actions but letting the user specify a tree structure with a grammar-like specification. These tools generate a class for each node in the tree in order to ensure valid tree construction. These approaches have in common that the grammar always requires being enriched with additional information about the details of the generated code. Our contribution in this paper is to provide a mapping to a grammar that is clean of any additional information but still carries enough information such that the essential information of a mapped Visitor is present. Our approach aims at using grammars as generated documentation for Visitors, but with properties, such that they might be interesting to investigate Visitors also from a language perspective.

An interesting combined approach that also shares several aspects with our mapping, is the use of a JavaCC grammar in the Java Tree Builder (JTB), originally developed by Jens Palsberg and Kevin Tao. A plain JavaCC grammar file serves as input to JTB, and from that grammar an object-oriented AST and its creation during parsing, following the design in [13], is generated. It also includes the generation of different depth-first Visitors. This provides a close relationship to our mapping, but in the opposite direction and with a different grammar design of the productions. It requires that the class hierarchy has one single root class and does not use inheritance from concrete classes. In the context of ASTs this is commonly considered a good design. Our approach aims at being applicable to the documentation of Visitors that operate on arbitrary data structures. Therefore our grammar can also represent the language of Visitors that are traversing across different class hierarchies, i.e. allowing to traverse Composite Patterns, and also inheritance from concrete classes can be represented. In particular, in our grammar the traversal order is explicitly defined.

5 Conclusions

We have presented a mapping from the Visitor Design Pattern to a formal grammar. The grammar is a context free grammar and its design directly reflects the essential aspect of the Visitor Pattern. It consists of only two kinds of productions. The chain productions correspond to the relevant inheritance relationships of the class hierarchy where accept methods of the Visitor Pattern exist. The second kind of productions represents the information, at which type of node a visit method exists (left-hand-side) and in which order the remaining object structure is traversed from that node type (right-hand-side).
The language generated by the grammar is the Visitor Language. A visit method invocation is represented by a terminal in the grammar. A word of the language represents one possible visit sequence. Thus, the set of all sequences of visit method invocations that a Visitor Pattern can perform on an object structure, is the Visitor Language. The *essential aspect* of the Visitor Pattern is the set of such sequences that it defines for an object structure. This essential aspect is represented by the Visitor Language.
Although it is well known that grammars can be used for engineering software systems, as is also discussed in the context of recent Grammarware work in [14], the application of design patterns is usually not understood as an implicit language definition. With the presented mapping from the Visitor Design Pattern to a formal grammar, we aim at making this correspondence more obvious and easier to recognize. The grammar can be used as precise documentation of an existing Visitor Design Pattern. This understanding may drag people, who are not used to using grammars, towards Grammarware. Therefore, the readability of the grammar is one of our main concerns.
This work is a contribution to understanding pattern design as language design, applied to the classic Visitor Pattern which can be found in many libraries today. We believe that a similar method can also be used for other Design Patterns, in particular those that incorporate the use of other patterns in some systematic way.
For the claim "Library Design is Language Design", and our adapted version, "Pattern Design is Language Design", we have presented a mapping from the Visitor Pattern to a grammar that generates such a language. Our hope is that this contribution adds to a broader acceptance of using grammars by software developers, beginning by using them for documenting Visitors, and that it may permit to recognize, understand, and investigate further the many languages that are implicitly defined in software systems.

Acknowledgements

I wish to thank Jens Knoop, Franz Puntigam, Daniel Quinlan, and the anony-

mous reviewers for valuable suggestions. This work was performed within the ARTIST2 Project IST-004527.

References

1. Andrew Koenig and Barbara Moo. *Ruminations on C++*. Addison-Wesley, 1996.
2. E. Gamma, R. Helm, R. Johnson, and V. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
3. Martin C. Carlisle and Ricky E. Sward. An automatic "visitor" generator for ada. *Ada Lett.*, XXII(3):42–47, 2002.
4. Joost Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 270–282, New York, NY, USA, 2001. ACM Press.
5. Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.
6. Johan Ovlinger and Mitchell Wand. A language for specifying recursive traversals of object structures. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 70–81, New York, NY, USA, 1999. ACM Press.
7. Arie van Deursen and Joost Visser. Source model analysis using the jjtraveler visitor combinator framework. *Softw. Pract. Exper.*, 34(14):1345–1379, 2004.
8. Norman Neff. Attribute based compiler implemented using visitor pattern. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 130–134, New York, NY, USA, 2004. ACM Press.
9. Andy Bulka. Design pattern automation. volume 13 of *Conferences in Research and Practice in Information Technology*, Melbourne, Australia, 2003. Australian Computer Society. Pattern Languages of Programs 2002. Revised papers from the Third Asia-Pacific Conference on Pattern Languages of Programs, (KoalaPLoP 2002).
10. Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and aspectJ. In *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, pages 161–173, Seattle, WA, November 2002. ACM Press.
11. Aino Cornils and Görel Hedin. Statically checked documentation with design patterns. In *33rd International Conference on Technology of Object-Oriented Languages, TOOLS Europe 2000*, pages 419–430. IEEE Press, June 2000.
12. Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *Lecture Notes in Computer Science*, pages 482–496. Springer, 2000.
13. Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, 1998.
14. P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. *ACM TOSEM*, May30 2005. To appear; Online since July 2003, 47 pages.
15. Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965.
16. Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In Laszlo Böszörményi and Peter Schojer, editors,

JMLC'03: Joint Modular Languages Conference, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, August 2003.
17. Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
18. Tom Tourwe and Tom Mens. A declarative meta-programming approach to framework documentation. In *Proceedings of the Workshop on Declarative Meta Programming to Support Software Development (ASE'02)*, July 19 2002.
19. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.
20. K. Koskimies. Object Orientation in Attribute Grammars. In H. Alblas and B. Melichar, editors, *Proc. International Summer School on Attribute grammars, Applications and Systems (SAGA'91), Prague, Czechoslovakia*, volume 545 of *LNCS*, pages 297–329. Springer-Verlag, June 1991.
21. Josef Grosch. Object-Oriented Attribute Grammars. In E. Gelenbe A. E. Harmanci, editor, *Proceedings of the Fifth International Symposium on Computer and Information Sciences (ISCIS V)*, pages 807–816, Cappadocia, Nevsehir, Turkey, October 1990.