Generic Visitors in C++

Nicolas Tisserand

Technical Report n°0312 - 20th October 2003

The visitor design pattern is a well-known software engineering technique that solves the double dispatch problem and allows decoupling of two inter-dependent hierarchies. Unfortunately, when used on hierarchies of Composites, such as abstract syntax trees, it presents two major drawbacks: target hierarchy dependence and the mixing of traversal and behavioral code.

CWI’s visitor combinators are a seducing solution to these problems. However, their use is limited to specific ‘combinators aware’ hierarchies.

We present here Visitors, our attempt to build a generic, efficient C++ visitor combinators library that can be used on any standard ‘visitable’ target hierarchies, without being intrusive on their codes.

Keywords
visitors, visitor combinators, C++, meta-programming, expression templates
Copying this document

Copyright © 2003 LRDE.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is provided in the file COPYING.DOC.
## Contents

1 Introduction 6

2 Visitor Designs 8
   2.1 First attempt .................................................. 8
   2.2 Multi-methods .................................................. 9
      2.2.1 A simple example ....................................... 9
      2.2.2 External single dispatch ................................ 9
      2.2.3 Double dispatch ......................................... 10
   2.3 The visitor design pattern .................................. 10
   2.4 Visitor combinators ......................................... 12
      2.4.1 Combination ............................................... 13
      2.4.2 Traversal combinators ................................ 13
      2.4.3 Conditional combinators ............................... 14
   2.5 What we want ................................................ 14

3 Implementation Techniques 16
   3.1 Generic combinators ........................................ 16
   3.2 Virtual adapter ............................................... 17
   3.3 Hierarchy Unrolling ......................................... 19
   3.4 Acting as Vanilla visitor .................................. 20
   3.5 Visitors versus Combinators ............................... 21
      3.5.1 A necessary distinction ................................ 21
      3.5.2 On exceptions ............................................. 22
   3.6 Static composition .......................................... 23
   3.7 Self referencing combinator ............................... 24
      3.7.1 Self referencing types ................................. 24
      3.7.2 The combinator proxy ............................... 25
      3.7.3 Benefits .................................................. 25
   3.8 Traversal ....................................................... 27
      3.8.1 Node type mapping ..................................... 27
      3.8.2 Subtree access .......................................... 27

4 The Visitors library 29
   4.1 Presentation .................................................. 29
      4.1.1 Introducing visitors ................................... 29
      4.1.2 What you get ............................................. 30
   4.2 Tiger use case ............................................... 30
      4.2.1 Writing adapting code ................................. 31
Chapter 1

Introduction

This report is about generic tree traversal and template meta-programming. Tree-oriented data structures are very commonly used in computer science. In the particular field of language processing and compilation, abstract syntax trees are used to represent a structured form of any arbitrary concrete input term in a given language.

An abstract syntax tree consists basically in nodes containing zero or more child nodes, non-terminals being the parent nodes and terminals being leaves. Figure 1.1 shows an example abstract syntax tree of the arithmetic expression $5*10+1$.

Since the whole tree is constructed through specific production rules, it is bound to the grammar which describes the term it was derivated from.

![Simple abstract syntax tree](image)

After their construction, trees can be walked through and processed.

There can be many ways of walking a tree. The most known traversal paths are the infix, prefix and postfix ones. Since traversal depends mainly on the nodes arities, given good descriptions of the nodes of the tree, it may be possible to define generic traversals (see section 3.8).

While walking the tree, treatments like printing the tree, type-checking, or optimizing can be applied to the nodes. Many of them heavily depend on the targeted tree nodes, but some, like dumping the tree could be written generically.

$^{1}$In the remaining of the report, we will use this simple arithmetic expression abstract syntax tree to illustrate most of our examples.
So, we have:

- Different grammars which produce different abstract syntax trees
- Different processings applied on the trees’ nodes
- Traversals that look the same, as long as we deal with tree-like data structures

While developing tree processing code, one would like to write only the abstract syntax tree specific treatments, and select an appropriate (either predefined or customized) generic traversal.

This is the goal of Visitors, our debut-library of generic visitors in C++. It has been developed since January 2003 as my research project at the LRDE. Its main goal was to use visitor combinators on the abstract syntax tree of the LRDE Tiger compiler (see Demaille; Appel, 1998).

Most of the non-C++ ideas come from the work work of Joost Visser on visitor combination and traversal control (see Visser, 2001).

This report is made of four parts. It introduces first the existing designs and the motivations for having designed Visitors, a generic visitors library in C++. On a second part, it focuses on several meta-programming techniques used in this library. Next, an example application of Visitors to the LRDE Tiger compiler is shown. The last part concludes with the limitations of the current framework, and tries to point out some future directions.
Chapter 2

Visitor Designs

This chapter deals with the existing tree traversal designs. It progressively shows several solutions to the tree traversal and processing problem and finishes by a description of the features that we would like our framework to offer. The designs presented here can be implemented in any object-oriented language featuring virtual methods: they do not use C++ specific parts. They are all already -more or less- known techniques.

2.1 First attempt

Using an object-oriented language, the first step towards developing tree processing would be to use encapsulation and group together in a same class, both the data and the operations performed on them. This seems to be a convenient approach and leads to the composite design pattern, as illustrated by figure 2.1.

![Figure 2.1: Naive composite design](image)

The composite, used "as is", may be well suited for small scale — write once, never modify—developments. However this design cannot be extended since adding one operation implies modifying the original class; nor reused because all the operations are bound to the hierarchy on
which they operate. Moreover, the traversal code is defined in each subclass of the hierarchy.
(In our example, the print and in the eval methods both contain the same traversal code.)

Adding some extensibility could be achieved by separating the data from the treatments. This particular need can be addressed through the use of multi-methods.

2.2 Multi-methods

Multi-methods are a generalization of virtual methods. Whereas the C++ virtual feature allows dispatch of a method call according to the dynamic type of one object (the first hidden this argument of the call), multi-methods allow dispatch on an arbitrary number of any of their arguments. This is also known as multiple dispatch.

Mainstream object oriented languages (like C++ or Java), do not feature multi-methods. In fact, the only multi-methods implementations I know are in CLOS (the Common Lisp Object System) or the upcoming Perl 6 language. There exist however some tricks to emulate them (see Alexandrescu, 2001), but they are more hacks than viable solutions. Despite these drawbacks, understanding them is a good way to define more precisely our needs.

2.2.1 A simple example

The following code snippet, written in an imaginary “multi-method-powered C++”, shows an example use of multi-methods. In this example, the call to operation() is dispatched to the adequate overloaded definition according to the dynamic type of both of its arguments.

```c++
void operation(virtual ConcreteType1*, virtual ConcreteAnotherType1*)
{ /* ... */ }
void operation(virtual ConcreteType2*, virtual ConcreteAnotherType2*)
{ /* ... */ }
Type* t = new ConcreteType1;
AnotherType* at = new ConcreteAnotherType1;
operation (t, at); // Calls the first method.
```

In this code, the ConcreteType1, ConcreteType2, ConcreteAnotherType1 and ConcreteAnotherType2 classes extend the Type class. The first method is called, even if t and at have a base class static type. Without multi-methods, legacy C++ method dispatch would not be able to resolve an appropriate method call.

2.2.2 External single dispatch

We can attempt to figure out how multi-methods could be applied to tree traversal needs. As a basic use case, multi-methods can be used to add methods to a class, outside of its definition. This “outside of class” single dispatch is shown in the piece of still imaginary C++ code below.

```c++
void print (virtual IntExp*) { /* ... */ }
void print (virtual OpExp*) { /* ... */ }
```
2.3 The visitor design pattern

2.2.3 Double dispatch

We might need more than a couple of isolated methods and would like therefore a full blown hierarchy of treatments. This could help to factor out code by inheritance. The following code shows an example of how such a hierarchy of treatments could look like in imaginary C++.

```cpp
struct Operation
{
    virtual void operation (virtual IntExp*) = 0;
    virtual void operation (virtual OpExp*) = 0;
};

struct Print : public Operation
{
    virtual void operation (virtual IntExp*) { /* ... */ }
    virtual void operation (virtual OpExp*) { /* ... */ }
};

Exp* e = new IntExp(51);
Operation* o = new Print;
o.operation (e);
```

In that particular case, double dispatch is used: the first made against the Operation dynamic type, and the second against the node dynamic type.

There is no need for a dispatch on \( n \) arguments (with \( n \) an arbitrary integer), so multiple dispatch is overkill. That’s a chance, because there exist a nice and convenient way to emulate the double dispatch in single dispatch languages: the visitor design pattern.

2.3 The visitor design pattern

From the design patterns book (see Gamma et al., 1994), the visitor design pattern intent is to “represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”

It encapsulates all the treatments to be performed on the nodes of a given tree on single classes, which extend a visitor abstract class. This top-level class contains one visit method for each tree node type (see figure 2.2). These visit method always take a tree node as a first and unique argument, and perform specific actions on it.

On the target side, each tree node define a special accept method. This one takes as first and unique argument a visitor and always perform the same operation: visiting itself, that is to say, call the visit method of the visitor with this as argument (see figure 2.3).
The class diagram of the visitor model for our simple arithmetic expressions is shown on figure 2.2.

While looking complicated, the visitor design pattern comes from a simple idea: “Double dispatch is not available ... so let’s do simple dispatch twice”.

This is the key concept, which is illustrated by the sequence diagram shown in figure 2.3, representing two accept(PrintVisitor) calls, performed first on an OpExp, and next on an IntExp. On each call, two dispatches are made: one according first to the node type, and then one according to the operation performed.
The visitor design pattern is also called “vanilla visitor”. In its original implementation, distinct (argument type dependent) names are used for the visit methods. In the examples shown above, as well as in the remainder of this report and inside the Visitors library, we rely on method overloading for the visit method. And this is—as seen later on this report—a very necessary point.

The visitor design pattern is a major step towards re-usability. It puts a clear separation between data and treatments and permits to factor out default-behavior code by inheritance. However, it is not generic enough: accept methods are bound to a specific abstract visitor class and visit methods are specific to the target hierarchy.

Worse, it forbids traversal control: traversal code is mixed with behavioral code inside each visit method. Despite the fact that a default traversal can be reused through inheritance, we still have to figure out how to traverse a given node every time we write a visit method for it.

Since we definitely do not want to write traversal code each time we write a visitor of a same hierarchy, we have to find something else.

### 2.4 Visitor combinators

Visitor combinators come from another simple idea: since monolithic visitors are not convenient enough (no genericity, no traversal control), we divide them in small atomic components. These small components can then be composed between them by visitor combinators to get the final visitor. This way, visitor combinators act like functions that take zero or more visitors and returns another visitor. This concept comes from Joost Visser’s paper on visitor combination and traversal control (see Visser, 2001) who takes its inspiration in the strategic term rewriting language Stratego (see Lämmel et al., 2002; Visser, 2002).
2.4.1 Combination

For example, an imaginary Compile visitor which performs successively escape checking, type checking and code translation can be made by combining the Escape, TypeCheck and Translate visitors, using the Sequence combinator:

\[
\text{Compile} = \text{Sequence} (\text{Sequence} (\text{Escape}, \text{TypeCheck}), \text{Translate})
\]

In this code, the Sequence combinator has the following prototype:

\[
\text{Sequence} : \text{visitor} \times \text{visitor} \rightarrow \text{visitor}
\]

This combinator simply combines two visitors together and returns a visitor that applies them sequentially. It can be easily implemented in C++, as seen in the following code:

```
struct Sequence : public ExpVisitor
{
    Sequence(ExpVisitor& first, ExpVisitor& second)
    : first_(first), second_(second) {}

    virtual void visit(OpExp& o)
    { first_.visit(o); second_.visit(o); }

    virtual void visit(IntExp& i)
    { first_.visit(i); second_.visit(i); }

    ExpVisitor& first_;
    ExpVisitor& second_;
};
```

This visitor combinator stores references on two other visitors and forwards sequentially to the first and then to the second the visit method calls.

Sequencing visitors is only the first step towards a visitor combinator framework: we still need to express tree traversal by visitor combination. For that purpose, traversal combinators are used.

2.4.2 Traversal combinators

The simplest kind of node traversal is immediate subtree traversal. This is handled by a special combinator called All, which takes a visitor and applies it to all subtrees of the current node when its visit method is called.

This way, the identity traversal, —which does nothing but traversing the whole tree— can be defined:

\[
\text{traversal} = \text{all}(\text{traversal})
\]
2.5 What we want

This recursive traversal-only combinator can be extended in order to apply some processings to the nodes it traverses. This is done by combining it with the Sequence combinator, and results in the \texttt{bottomup} and \texttt{topdown} combinators defined below.

\begin{align*}
\text{topdown}(v) &= \text{sequence}(v, \text{all(topdown}(v))) \\
\text{bottomup}(v) &= \text{sequence}(\text{all(bottomup}(v)), v)
\end{align*}

The first one simply applies the visitor \( v \) first and next continues the traversal by recursive calls. The second makes the traversal first and then applies the visitor. They respectively define prefix and postfix paths. Those combinators alone are not sufficient to encode complex traversals: they only define full tree paths and there is no way to cut the traversal. This is the goal of conditional combinators.

2.4.3 Conditional combinators

Conditional combinators enable the concept of visit success and failure: any combinator must either fail or succeed. In the original article (see Visser, 2001), conditional combinators are implemented by using exceptions\(^1\). A failing combinator throws a VisitFailure exception, while a successful one do not throw anything.

The \texttt{fail} combinator can be defined: this basic combinator simply fails without doing anything more.

This new feature modifies the semantics of our previous Sequence combinator, which now behaves like a boolean \texttt{and} because it can be broken by an uncaught VisitFailure exception. If the first visit fails, the second will never be made.

Given an “and-like” combinator, we can define an “or-like” one too. It is called \texttt{Choice}. \texttt{Choice(v1, v2)} applies \( v1 \) first and \( v2 \) only if \( v1 \) fails. Other conditional combinators such as \texttt{One} or \texttt{Try} can be defined too, and complex traversal can be expressed, only with those basic combinators.

2.5 What we want

Visitor combination is a very nice and versatile concept. It clearly separates the traversal code from behavioral code. Given a specific target tree hierarchy, re-usability is greatly improved: we just have to write the behavioral code, and compose it with traversal combinators.

However, those traversal combinators are still bound to a specific hierarchy and thus must be written once for every target hierarchy. There already exist systems that avoid this annoying point (see Visser, 2001), but they present two major drawbacks:

- They rely heavily on dynamic polymorphism, which uses dynamic method binding and thus can lead to performance issues in critical loops (see Duret-Lutz et al., 2001).

- They can only be used on specifically crafted hierarchies (other than the vanilla visitor design pattern) where node classes inherit from a framework abstract class.

\(^1\)This is however not the case for the \texttt{Visitors} library (See 3.5.2).
Thus, in order to enable visitor combination on a given hierarchy, one of the two following approaches must be chosen:

- Rewrite a whole combinator framework for each specific hierarchy. This leads to code duplication and time waste.
- Make modifications to the existing target hierarchy in order to let it accept a given combinator framework. This is too intrusive, and may even not be possible, when dealing with commercial libraries.

None of these two solutions seem acceptable. Visitors, our C++ generic combinator framework, uses a third one: design a generic visitor combination framework that can be used “as is” by all the hierarchies who implement the overloaded vanilla visitor interface. This was made possible by using C++ meta-programming techniques.
Chapter 3

Implementation Techniques

This part deals with several C++ techniques that were used in the implementation of our Visitors library. They are mostly meta-programming related.

Some of these techniques —like the virtual adapter— are a variation over a well known pattern or technique, while some others —like hierarchy unrolling— are far less common, but still useful in our case. They are almost all contributing to the same goal: building various visitors for any target hierarchy just by writing a small amount of code and still remaining efficient.

3.1 Generic combinators

Many combinators are not bound to a specific target hierarchy. For instance, Identity succeeds without doing anything with every node of any hierarchy. Similarly, Sequence always applies its first combinator and (if this succeeds) its second just after, to every node of any hierarchy.

Implementing those visitor combinators once for all would be nice. Using C++ static genericity is one of the natural ideas that comes to mind with such a desire.

The following code shows how a generic Identity combinator could look like.

```cpp
template < class AbstractVisitor >
struct Identity : public AbstractVisitor
{
    template < typename T >
    void visit (T& t) {}
};
```

In this code, a template Identity class is defined. This class, in order to behave like a regular visitor class must fulfill two roles:

- Extend an abstract visitor class
- Define a generic virtual visit method

For that purpose, Identity is parameterized by AbstractVisitor (the base class of the visitor hierarchy) and inherits from this parameter. Next, a generic virtual visit method is defined.

This is where problems arise: the template visit method defined inside the class is not a virtual method definition. This code does not compile, because in C++ operation polymorphism
(run-time polymorphism) and parametric polymorphism (compile-time polymorphism) cannot be mixed this way.

If our implementation language featured a kind of virtual template construction, the solution to this problem would have been trivial. But that is unfortunately not the case. We must find another implementation to bridge the gap between static and dynamic polymorphism. We need some kind of virtual adapter.

### 3.2 Virtual adapter

The virtual adapter is an application of the object adapter (also known as wrapper) design pattern (see Gamma et al., 1994).

![The object adapter design pattern](image)

The standard object adapter model is shown in figure 3.1. Its primary purpose is to adapt incompatible interfaces by using simple delegation. For that purpose, it aggregates one instance of the adaptee and forwards request calls to it.

Our solution slightly differs from the original pattern since it substitutes a one-to-one delegation to a many-to-one delegation: the virtual adapter goal is to allow a generic (template) definition of similar overloaded virtual methods. This is done by delegating in each virtual method a call to the same generic method. The following code shows a minimal implementation of the virtual adapter.

```cpp
struct Abstract // Abstract class
{
  virtual void virtual_method (int) = 0;
  virtual void virtual_method (std :: string) = 0;
};
```
### 3.2 Virtual adapter

```cpp
struct ConcreteGeneric
{  // Generic concrete class

  template < typename T >
  void method (T& t) { /* ... */ }
};

struct Concrete : public Abstract  // Virtual Adapter
{

  virtual void virtual_method (int i)
  { concrete_generic.method (i); }

  virtual void virtual_method (std::string s)
  { concrete_generic.method (s); }

  ConcreteGeneric concrete_generic;
};

int main ()
{
  Concrete concrete;
  concrete.virtual_method (51);
  concrete.virtual_method ("Imhotep");
}
```

We have here an `Abstract` abstract class which declares two overloaded virtual methods that we want to define generically. The `Concrete` class extends the `Abstract` class and defines virtual methods. These virtual methods contain both a delegation to a generic method implemented in the class `ConcreteGeneric`. This delegation is made by calling the `ConcreteGeneric::method` method on the aggregated `concrete_generic` object.

In the `Visitors` library, the delegation is made either through static genericity (`ConcreteGeneric` is a template parameter) or through dynamic genericity (`concrete_generic` is a pointer to an abstract class). In the first case, inlining is made possible and delegation occurs with no runtime cost with good optimizing compilers.

The main drawback of this adapter is that it forces one to define all the `virtual_method` methods in the `Concrete` class, although they all share the same code. This can be considered harmless in our simple example, but in the context of a large hierarchy it makes sense to look for a way to automate such declarations. Moreover, in our case, we cannot fall back into defining such methods by hand, because we want our framework to be able to adapt itself to foreign hierarchies (e.g., with an arbitrary number of visit methods defined).

Therefore, we need some kind of “automatic way” to declare methods. This is the purpose of the “hierarchy unrolling” trick.

---

1In this implementation, it could seem quite useless to store a `ConcreteGeneric` instance inside the `Concrete` class: we could have made the `ConcreteGeneric::method` either static inside its class or inside the `Concrete` class or completely external, since `ConcreteGeneric` does not store any state. But that is not the case in the `Visitors` library.
3.3 Hierarchy Unrolling

Hierarchy unrolling is a C++ meta-programming trick that allows one to generate a whole hierarchy from just one template instantiation. It is yet another variation on the Barton & Nackmann trick (see Barton and Nackman, 1994; Veldhuizen, 1999), where a concrete class is parameterized by a list of classes composing the final hierarchy. This can be used to define a set of similar overloaded methods.

An example implementation is shown in the following code, where a generic Print class is defined and then instantiated with a list containing two type elements (int and std::string) as template parameters. The resulting statically instantiated class contains two print methods: one for each type.

```cpp
#include <iostream>
#include <string>

struct EmptyList {};

template <typename Head, typename Tail = EmptyList >
struct List {};

template <typename List>
struct Print {};

template <typename Head, typename Tail = EmptyList > // Lower classes specialization
struct Print < List < Head, Tail > > : public Print < Tail >
{
    using Print < Tail > :: print;
    void print (Head h) { std :: cout << h << std :: endl; }
};

template <typename Last>
// Top class specialization
struct Print < List < Last > >
{
    void print (Last l) { std :: cout << l << std :: endl; }
};

int main ()
{
    Print < List < int, List < std :: string > > > printer;
    printer.print (51);
    printer.print ("imhotep");
}
```

This code fragment may at first seem quite obsfucated. It is just a weird way to inform our friendly C++ compiler that we want it to iterate over a list of types and construct a class which contains methods defined for each of these types.

This is done by using static lists (see Alexandrescu, 2001). Static lists are empty classes parameterized by two classes: the head of the list, which can contain any type, and its tail which contains the remainder of the list (another List type).
Iteration over the list is done through inheritance. Each iteration step adds one level in the final hierarchy. At each level, a new specialization of the `Print` class is instantiated and defines a new `print` method.

The last element specialization defines the top level class, while the more general `List < Head, Tail >` specialization defines the lower classes.

The resulting statically instantiated class defines a method for each type of the list, and for each instance of this class, a unique virtual table is built, and there is no additional indirection step involved, which means no additional runtime cost between objects instantiated from unrolled hierarchies and from the corresponding hand-written monolithic classes. An example of such an unrolled hierarchy\(^2\) is shown in figure 3.2.

Figure 3.2: An example unrolled hierarchy

### 3.4 Acting as Vanilla visitor

Hierarchy unrolling can let our C++ compiler instantiate a class with an arbitrary number of similar methods. So we can write a generic class than generates a concrete vanilla visitor class for any target visit-able hierarchy.

Generating such classes, requires:

- The abstract visitor type (because our class must inherit from it, in order to be accepted by a target node).

- The list of visited types (because we must implement all the visit methods in order to have an instantiable class).

Both of the former items can be expressed as types. (The first one is already a type, and the second can be written as a static list). So we may be able write a generic class which is parameterized by these two types and generates -once instantiated by the C++ compiler- the following class.

\(^2\)Hierarchy unrolling can be used for several other purposes. It was first used on a project where I needed a generic variant class: a class which could act as a “smart union” storage class. Such a class can be trivially written by hand for a specific set data types, but writing this class generically (statically parameterized by the set of data types) isn’t that easy. The hierarchy unrolling trick makes it possible.
Indeed, this can be achieved by using the hierarchy unrolling trick, as seen below.

```cpp
struct Visitor : public AbstractVisitor {
    virtual void visit (Type1&) { /* ... */ }
    virtual void visit (Type2&) { /* ... */ }
    // ...
    virtual void visit (TypeN&) { /* ... */ }
};
```

This `Visitor` class differs from the former simple hierarchy unrolling example just by one extra feature: the `AbstractVisitor` template parameter. It is the abstract visitor type which is accepted by the `accept` methods of the target hierarchy. The last element class specialization inherits from it, so that the whole unrolled hierarchy inherits from it too, making the bottom class an instantiable `AbstractVisitor` sub-type.

### 3.5 Visitors versus Combinators

Before going any further in this report, there is an important vocabulary point in the `Visitors` framework that must be emphasized: the distinction that is made between a visitor and a combinator.

#### 3.5.1 A necessary distinction

In short, a combinator cannot be directly accepted by a target hierarchy, whereas a visitor can. This distinction is essential for several reasons.

- Static composition forbids direct acceptance of a combinator by a target hierarchy. Indeed there is neither an abstract class, nor a hierarchy for combinators (because of the static
dispatch), so there is no chance that a combinator will ever implement a vanilla visitor interface.

- The interface of the combinator visit methods differ from the visitor ones, since visitor success or failure is a concept that comes from visitor combination and is absolutely not a part of the vanilla visitor design pattern.

### 3.5.2 On exceptions

On the original implementation of visitor combinators in Java (see Visser, 2001) the conditional failure/success mechanism of the combinators was implemented using exceptions. This approach presents several drawbacks that prevented it from being used in Visitors too.

Exceptions are part of the C++ core language and are very useful to break execution flow on exceptional cases. Because of this breaking behavior, their internal implementation often do not allow efficient optimizations to be performed by the compiler. Moreover, the C++ throw mechanism itself is slow on many 3 implementations and therefore exceptions should not be used as an alternative return mechanism.

In our particular case, there is another drawback in using exceptions: since our framework adapts itself to foreign hierarchies, using exceptions in our visitors would mean propagating exceptions through foreign code that is very unlikely to support handling them properly. In short: we do not want to break the flow of the adapted code. Another mechanism must be found.

In the original implementation, exceptions are only used to specify success or failure. There is only one type of failure exception, and throwing it means failure, while not throwing anything means success. This mechanism can be replaced by a more conventional value returning one: make visit_ 4 return a boolean value. Returning from the method with a true result would mean success, and with a false result, failure.

This works well, but it forces the user to always return the success/failure status of a visit_ method. Some combinators become a little more tricky too, but the overall result seems natural. The following code snippet show the differences between two versions of the sequence combinator: one using exceptions, and the other using return values.

```cpp
// Without exceptions.
template <class T>
bool visit_ (T& t) {
  return first.visit_ (t) && second.visit_ (t);
}

// With exceptions.
template <class T>
void visit_ (T& t) {
  first.visit_ (t);
  second.visit (t);
}
```

In the first version, visit return status is explicitly specified and handled. In the second form, the two visit_ calls potentially throw visit failure exceptions.

3If not all...

4I gave this name to the visit methods of the combinators, in order to avoid confusion and tricky overloading in virtual adapters.
There is an additional problem regarding this approach: we stopped exception intrusion into client code, but we still need a way to communicate failure or success status between combinators who executes client code. Those problematic combinators are mainly traversal ones: they are the frontier between framework and client code. Thus, the visit methods they contain call accept methods defined in the nodes of the target hierarchy.

Our solution is to store the combinator\(^5\) and the boolean value of the current state of the visit inside a freshly created visitor\(^6\). Next, this visitor gets accepted by a node, where its return status will eventually be modified by other combinators\(^7\). The status is finally returned after the accept call. Hence, the client code remains isolated.

### 3.6 Static composition

Being able to write visitors and combinators for any target hierarchy, the next step is to make it effectively enable visitor combination.

At a first glance, an implementation similar to the one used in the original paper (see Visser, 2001) could be used. This leads to the code shown below which implements a generic Choice combinator.

```
template < typename AbstractCombinator >
struct Choice {
   Choice (AbstractCombinator& first, AbstractCombinator& second)
    : first_(first), second_(second) {} 

   template < typename T >
   inline bool visit_ (T& t)
   { return first_.visit_ (t) || second_.visit_ (t); }

   AbstractCombinator& first_;
   AbstractCombinator& second_;
};
```

In this code excerpt, the template parameter AbstractCombinator is an abstract class, so the two method calls inside the generic visit method both involve a dynamic dispatch. This is highly unwanted, since combinators might be rather big, the overhead of a large number of dispatches could be far from negligible (see Duret-Lutz et al., 2001).

With efficiency in mind, the idea that appears to be a better implementation of combination is the use of static composition “à la” expression templates (see Veldhuizen, 1999). This well-known optimization technique relies on static binding and thus allows the compiler to massively inline method calls, potentially leading to very efficient code. The idea is the same as usual in meta-programming: all the exact types must be known at compile time. This is the case in the new Choice implementation shown below.

```
template < typename First, typename Second >
struct Choice {
```

\(^5\) A combinator cannot be accepted “as is” by a target node.

\(^6\) See section 3.7.2 for more information about this combinator proxy class.

\(^7\) In All(V) for instance, V will modify the return status of All.
### 3.7 Self referencing combinator

Choice (First& first, Second& second)
  : first_(first), second_(second)
{}  

template < typename T >
inline bool visit_ (T& t)
{ return first_.visit_ (t) || second_.visit_ (t); }

First& first_;  
Second& second_;  
};

This Choice combinator can then be instantiated as in the following example code.

FirstCombinator first_combinator;  
SecondCombinator second_combinator;  
Choice <FirstCombinator, SecondCombinator>
  my_choice (first_combinator, second_combinator);

As in regular expressions templates, overloaded operators and/or convenience functions can be easily written to ease the use of such static combinators on the right hand side of a combinator object instantiation.

But this new composition scheme has a drawback: it is a bit less user friendly, because it forces the user who instantiates a combined object to write the full name of its whole type on the left hand side of the assignment expression, which can be pretty long and cumbersome.

#### 3.7 Self referencing combinator

Many traversal combinators are referencing themselves in their own definition. This unfortunately breaks our static composition scheme.

##### 3.7.1 Self referencing types

Using static composition to build the following combinator is impossible:

traversal = all(traversal)

The C++ type of such a combinator should be All < All < All < All ... (an infinite type) but this is not possible and does not make sense.

Self referencing types are impossible in C++. While this problem is harmless in the original context of expressions templates, it is very annoying in our case. So we need a proper way to break static composition.

---

8 Arithmetic expressions are trees where sub-nodes never refer to upper ones.
3.7.2 The combinator proxy

Our solution uses a concept coming from the Spirit C++ library (see Spirit, 2001): a static combinator proxy which enables static composition breakage by inserting an extra dynamic dispatch.

As quoted from the design pattern book (see Gamma et al., 1994): “An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject.”

We use in Visitors a proxy that provides the same interface as a combinator: it defines non-virtual (generic or specific) visit_ methods. Its added-value relies in the instantiation method: the proxy is not bound to a specific combinator as soon as it is created. Instead, this is postponed to a further invocation of the generic operator=. This delay involves one constraint: an empty pointer has to be stored in the proxy class. Since the type of the pointed object cannot be a known at compile time, it has to be an abstract type. But combinators are statically combined, so there is no such top level class :

The solution used is to implement an additional hierarchy of dynamic combinators. This is illustrated in the code snippet below.

```cpp
struct Combinator {
    template < typename V >
    Combinator& operator= (V& v) {
        v_ = new ConcreteCombinator < V > (v);
        return *this;
    }

    AbstractCombinator* v_; // static visit methods are delegated to v_
};
```

Here, the Combinator class is a non-template class which delegates the visit_ methods (not shown here for simplicity) to v_. The static type of v_ is AbstractCombinator and its dynamic type is ConcreteCombinator < V >, where V is the the type of the real subject (the “proxified” combinator).

ConcreteCombinator is an intermediate dynamic to static adapter. It extends the AbstractCombinator class and works like the virtual adapter shown in the section 3.2, but reversed.

When a call like v_->visit is made inside a Combinator visit_ method, a dynamic dispatch is made first, according to the dynamic type of v_, and then a call to the statically known visit_ method of V is made inside the template ConcreteCombinator method. This way, a template class is wrapped inside a non-template class which offers the same interface.

3.7.3 Benefits

This Combinator class is very handy. Its instances, when used as the left hand side of a complex visitor combinator assignment provides a triple benefit:

- The exact type of the right hand side of the expression no longer has to be written in full.
- Self referencing combinators are now made possible.
3.7 Self referencing combinator

- Successive assignments of the same Combinator instance are possible.

**Handy assignation**

Combinator is non-template, so one can write:

```cpp
Combinator v = all ( v1 && v2 )
```

instead of

```cpp
All < Sequence < V1, V2 > > v = all ( v1 && v2 )
```

A similar effect could be achieved by using the non-standard `typeof` C++ extension:

```cpp
typeof(all ( v1 && v2 )) v = all ( v1 && v2 )
```

But it’s non-standard and the right hand side of the expression has to be written twice.

**Self referencing combinators**

The Combinator class defined here can perfectly be statically combined with other combina-
tors. Moreover, since it is non-template, it can be safely self-referencing, like in the following
statement:

```cpp
Combinator top_down_v = v && all(top_down_v)
```

In C++, it is perfectly valid to take a reference to an object inside its own initialization (see
standard, 1998), so self-referencing works here like expected! The right hand side of this as-
signment type is `Sequence < V, All < Combinator > >`, which is a finite type.

**Successive assignments**

The Combinator class constructor does not take any argument. It can thus be instantiated once,
and, because of its dynamic behavior, be assigned several times by calling its `operator=` on
different successive other combinators. It acts like a combinator container, which is both handy
and natural.

The techniques presented until now allow generic implementation of a wide variety of effi-
cient generic combinators. But there still remain an essential technique: how to achieve generic
traversal.
3.8 Traversal

Traversal combinators are the most tricky of the combinators implemented in Visitors. Firstly, because they are the only ones who execute client code (the accept methods) and thus “leave” the framework code. Secondly because they are the only one who need to know how to traverse the target hierarchy.

The main direction followed during their development was to forbid intrusion in client code and to find a way to adapt foreign hierarchies.

Since our primary goal was to use visitors on the existing LRDE Tiger compiler implementation—which is used as teaching material—being intrusive on this implementation and equip it with our experimental code was absolutely not an option. We had to externally describe the target hierarchy.

3.8.1 Node type mapping

In order to describe a target hierarchy, node kinds have to be defined. Three categories are distinguished:

- Leaf nodes, which never contain subtrees.
- Nary nodes, which contain a compile-time known number of subtrees.
- List-like nodes, which contain a dynamic number of subtrees.

Given those distinctions, we have to find a convenient way to statically map them to hierarchy nodes.

Static type mapping is possible by using traits (see Veldhuizen, 1999). Basically, a VisitedTypes empty template class is defined. It is parameterized by a type, and by a target hierarchy. This class has to be partially specialized for each target node type of the hierarchy. The traits specializations must contain a special typedef, which defines the node kind, and special members, which allow subtree access.

3.8.2 Subtree access

Because writing such members subtree accessors is cumbersome, some convenience classes are provided by the framework to permit easy writing of them.

They take advantage from the fact that often, in nary target nodes, there are already functions that return the accessed element. In that case, writing an accessors inside the VisitedTypes specialization is just a matter of making a delegation to it. A special template class, parameterized by a static list of the target hierarchy accessor method pointers can be defined. This class defines the VisitedTypes accessors which contains the delegating call. This call can be inlined by the compiler, since the method pointer is known at compile time (see Veldhuizen99techniques).

Writing a traversal combinator is like writing three variants of a combinator (one for each node kind). Indeed, a static dispatch must be made according to the node kind: the correct visit implementation must be found among the three available. This is done by making an extra template class (say CombinatorImpl), with three partial specializations (one for each node kind) containing a visit_impl method (the visit implementations). These methods are accessed from the main template visit_method of the combinator by instantiating CombinatorImpl < node_kind > where node_kind is the current node kind, depending on the parameter of
visit\_ method. This statically selects the correct implementation when this visit\_ template method is instantiated, inside the traversal combinator.

A complete example of hierarchy adapting is shown in section 4.2.1.
Chapter 4

The Visitors library

4.1 Presentation

This chapter is a quick guided tour of the Visitors library.

4.1.1 Introducing visitors

The Visitors library is a framework designed for C++ visitor combination. It allows non-intrusive adapting of foreign visitor-enabled tree-like hierarchies, for which it can produce visitors by combination.

It is available freely, under the terms of the GNU GPL at the LRDE website. (See the project homepage: http://www.lrde.epita.fr/Projects/Visitors)

Before using it, there is an important distinction to understand between the different kinds of source code that are involved in the complete system.

- Target code. This is the abstract syntax tree to work on. It is written once, possibly before even considering using the Visitors library. The only restriction about this code is that it must implement correctly the overloaded visitor design pattern.
- Framework (the Visitors library). It contains the hierarchy adapting system, and the generic combinators.
- Adapting code. This code is written once for each specific hierarchy and describes it (how to traverse it and what are the visited nodes.)
- Client specific code. This is the code written by the final user. It can be subdivided into the following categories:
  - Custom generic combinators, that can extend the whole framework and be used on any other hierarchy.
  - Custom (abstract syntax tree specific combinators) that can only be useful on a specific hierarchy.
  - Visitors instantiations and use: the final code that makes use of all the above.
4.2 Tiger use case

The target code and framework code “zones” are materialized in the class diagram shown in figure 4.1.

![Class Diagram](image)

**Figure 4.1: The Visitors classes**

4.1.2 What you get

The Visitors library is made of the following components:

- A visitor combinator framework.
  - The Visitor and ConcreteVisitor classes are classes whose instances can be directly accepted by a target node.
  - The Combinator and ConcreteCombinator classes are classes that can be combined with other visitor combinators.
- Hierarchy adapting tools.
- A set of predefined generic combinators. These ones can be refined in several categories:
  - basic combinators
  - conditional combinators
  - print combinators
  - traversal combinators

4.2 Tiger use case

One of the primary goals aimed by the Visitors library was to use its visitors on the LRDE Tiger (see Appel, 1998; Demaille) compiler abstract syntax tree.
4.2.1 Writing adapting code

The following snippets are taken from the Tiger abstract syntax tree adapting code.

```cpp
struct TigerHierarchy // Target hierarchy
{
    // Abstract visitor type
    typedef ast::Visitor visitor_type;

    // Types of the hierarchy
    typedef List<IntExp, List<OpExp>> types;
};
```

This first excerpt shows the `TigerHierarchy` class. This class is passed as a template parameter to every combinator which needs it in order to allow hierarchy unrolling and visit method generation. Thus it contains a `typedef` for the hierarchy abstract visitor type and another that defines the list of types to be visited. But this is not sufficient, since traversal combinators still need to get information on how to traverse nodes. This is the role of the `VisitedTypes` class, shown below.

```cpp
// OpExp
struct VisitedTypes<OpExp> : public NaryNode<OpExp,
    List<Accessor<OpExp, Exp&>,
        &OpExp::left_get>,
    List<Accessor<OpExp, Exp&>,
        &OpExp::right_get>>
{
};

// IntExp
DECLARE_LEAF_NODE(ast::IntExp);
```

That second excerpt is a traits specialization. It specifies how to traverse an `OpExp` node. It inherits from the `NaryNode` helper template class and thus gets correct accessors for its child nodes. `OpExp::left_get` and `OpExp::right_get` are the accessor method provided by the Tiger abstract syntax tree. They are automatically called by the code generated by the template instantiation of the `NaryNode` class.

After this essential code is written, combinators can be used on the target hierarchy.

4.2.2 Writing generic combinators

Generic combinators are rather easy to write, provided they do not perform traversal operations (in which case they need to discriminate on the node kinds). Two examples of such generic combinators are shown in the following sections.
The Match combinator

This combinator simply succeeds when it visits a node whose type is equal to its template parameter. It is included in Visits, and its implementation is very simple:

```cpp
template <typename Node>
struct Match : public Fail {
    using Fail::visit_

    bool visit_ (Node& node)
    {
        return true;
    }
}
```

We benefit here from the C++ overloading resolution model, which gives a higher precedence to the non-template overloaded methods. This way, when a visit method call is performed on a Match object, if the object’s type is Node, then the visit implementation above is called. In all other cases, the Fail implementation is called, which leads to the correct behavior.

The PrintType combinator

```cpp
struct PrintType {
    template <typename T>
    bool visit_ (T& t)
    {
        const std::type_info& ti = typeid(t);
        const char* type = ti.name();
        os_ << type;
        return true;
    }
}

static PrintType print_type;
```

This is a type printing combinator, whose generic visit method prints the type of the visited node. It uses C++ runtime type identification to get a mangled name for the class1. A prefix type-printer can then be defined by combination with a TopDown combinator, as shown below:

```cpp
fifty_one.accept(visitor(top_down(print_type & print("\n"))));
```

1Because name mangling is not standardized among C++ ABI implementations, the non-portable demangling code is not shown here.
In this code && is a convenience operator that returns an instance of the Sequence combinator. Similarly, print and top_down are convenience functions that return an instance of, respectively a print combinator and a TopDown combinator. Once accepted by the sample fifty_one Tiger abstract syntax tree tree, it prints the types in a prefix fashion, separated by carriage returns.

4.2.3 Writing specific Combinators

More advanced combinators can be written too. The following example shows how to build a combinator which figures out whether an expression is constant or not.

```cpp
Match<OpExp> op_match;
Match<IntExp> int_match;

Combinator<> is_const =
    int_match
    || (op_match && all(is_const));

Visitor<> is_const_visitor =
    (is_const
     && *new Print("const"))
    || *new Print("not_const");

exp.accept(is_const_visitor);
```

First, two instances of the Match generic combinator are defined. They both are used to match nodes: one for the constant leaf nodes (the IntExp nodes) and one for the operator. Next, a combinator is_const is defined. It specifies that in order to succeed, the is_const visitor must either be accepted by an IntExp or be accepted by an OpExp for which both subtrees have successfully accepted the is_const visitor. It is a declarative syntax, quite unusual in C++, but very appropriate in our case. After having a combinator that succeeds on constant nodes, we just have to find a way to output the result. That's the purpose of the is_const_visitor. This visitor prints "const" when accepted by a constant node—one for which the is_const combinator has succeeded—and prints "not const" otherwise.

---

2The visit_ method of this class prints the string with whom it was constructed
Chapter 5

Conclusion

5.1 Results

The Visitors library offers generic visitor combination to any visit-able tree-like hierarchy. It is well suited in the field of language processing and generic tree traversal.

Moreover, it is a satisfactory proof of concept on several points:

- Generic combinator definition
- Static combination
- Non intrusive adapting of foreign hierarchies

However, as a young research project, it still has some drawbacks and limits that could be worked on.

5.2 Applicability

5.2.1 The target hierarchy

Although being generic, the Visitors library may not be usable in some cases, because the target hierarchy must meet some requirements.

- There should be one accept method in each visited class.
- Nodes must behave like a list node or like a n-ary node or like a leaf node.
- The target hierarchy should define overloaded visit methods instead of visit_foo methods (as seen in the original design pattern).

In the LRDE Tiger compiler (see Demaille), there are some issues that prevent the use of the Visitors library on the whole abstract syntax tree.

- Some nodes behave simultaneously like a n-ary node and a list node. This is the case for the FunctionDec node, which defines two subtrees for its body and result, and the list of parameters, all inside the same class. This heavily breaks our traversal model. Moreover, linearly traversing such nodes does not make much sense.
• The interface of the visitors is not completely uniform: there is a visit method for `decs_t`\textsuperscript{1} in the `Visitors` abstract class, but no accept method in `decs_t`. This breaks all the traversal combinators of our framework, since in each of them, there is call to the accept method of the targeted node.

Those major problems cannot be fixed in the framework code, and thus would involve some rewriting in the target code. This emphasizes the limits of the `Visitors` library, which can only address “well-crafted” visitable hierarchies.

5.2.2 Usability

Visitors is an active library which unfortunately carries all the major disadvantages of that kind of software library:

• CPU stress: the compilation of a `Visitors` client program is a CPU intensive task, which relies heavily on the C++ compiler to generate efficient code and make compile-time computations such as loops and list browsing.

• Compiler stress: Due to the differences among C++ compilers implementations and the uncommon nature of the visitors code, compiler support is very poor: only g++ 3.0 or later can compile the `Visitors` library.

• Developer stress: On the library developer side, developing meta-C++ code is harassing and error prone. Many syntactic constructions are unnatural and using the compiler as a meta-code interpreter has the annoying side-effect of greatly obfuscating the source code.

• User stress: Since the library is just a set of headers, without any object code, the library user shares the pain of the library developer: it gets the same cryptic error messages, the same compilation times, and the same compiler compatibility issues. It however gets an extra annoyance, specific to the `visitors` library: the adaptation mechanism. Indeed, writing adapting code for complex hierarchies is very cumbersome. Hopefully, this has to be done just once.

5.3 Performance

Performance has been a major concern during the development of `Visitors`. While extensive tests have not yet been performed, we can consider the following example: a simple combinator, working on an implementation of an arithmetic expression abstract syntax tree:

\[
\text{Combinator} \leftrightarrow \text{c} = (\text{try\_ (int\_match)} \&\& \text{try\_ (binop\_match)}) \&\& \text{all (c)};
\]

When accepted by a small tree expression (2 operators and 3 numbers), 102 framework visit method calls are performed:

• 7 calls using dynamic dispatch

• 95 calls using static dispatch

\textsuperscript{1}This is a typedef for a list of declarations: std::list<Decs*>
So, more than 90% of the visit methods calls can be inlined by the compiler. This percentage is not so surprising: as stated sooner in this report, the only dynamic dispatches are made in traversal combinators, when accessing subtrees, and in adapters, when leaving the framework and executing client accept methods.

5.4 Future

5.4.1 Implementation

The static genericity mechanism used in the Visitors framework is loose: generic combinators can be parameterized by any types. This could be greatly improved to a more secure constrained genericity through the use of static concept checks, such as the ones used in the STL (see sgi) or in the Boost (see Siek and Lumsdaine, 2000; bcc) libraries.

Another lack in the field of C++ implementation concerns the constness of the framework: developing a const aware framework was initially not pointed out as a primary goal. Now that the interfaces are stable, pondering on it might be necessary.

5.4.2 Expressiveness

The expressiveness of the library combinator language is not wide enough: although traversal and matching can be easily expressed, there is still one gap that has to be filled: there is no generic node substitution mechanism, which can annoy the user and even render the framework unusable in some cases. Instead, a nice mechanism could be a generic support of “match and rewrite” constructions. This issue require some thought, however.

On the other hand, functional tools “à la” FC++ (see FC++, 2002) could be easily added to the existing framework. This could for instance allow one to define partially applied combinators or to make use of place-holders to define combinators, like in the following example definition of the top-down combinator.

\[
\text{Combinator top\_down} = \text{sequence } (_1, \text{all}(\text{top\_down}(\_1))))
\]

In this example, \_1 is a placeholder, and the resulting top\_down object behaves like a partially applied functor.

5.4.3 Code generators

A default abstract syntax tree and adapting code generator could also be implemented. It could for example take its grammar definitions from SDF (see de Jonge and Visser, 2001) specifications.

5.4.4 External bindings

Binding the library with other systems could be another future improvement. The Spirit parser framework might be the first candidate on the list. Since it provides a parser generator framework implemented using meta-programming techniques, generating Visitors compliant abstract
syntax tree classes from its parser definitions, and using generic visitor combinators on them could be studied.

5.4.5 Tiger

Instead of trying to adapt the abstract syntax tree hierarchy of the Tiger LRDE compiler, addressing the intermediate representation tree could be an interesting future work. Indeed, this latter hierarchy do not carry the previous drawbacks and thus could be successfully used with the Visitors library.


# List of Figures

1.1 Simple abstract syntax tree .............................................. 6
2.1 Naive composite design ................................................... 8
2.2 The visitor design pattern ............................................... 11
2.3 Visitor sequence diagram ................................................ 12
3.1 The object adapter design pattern ...................................... 17
3.2 An example unrolled hierarchy .......................................... 20
4.1 The Visitors classes ...................................................... 30
Index

abstract syntax tree, 6, 7, 29–31, 33–37
abstract syntax trees, 6, 7
adapter, 16–18, 22, 25, 36
C++, 7–10, 15, 16, 19, 20, 22, 24–26, 29, 32, 33, 35, 36
combination, 7, 12–14, 22, 23, 29, 32, 34
composition, 21, 23–25
dispatch, 1, 9–11, 22, 23, 25, 27, 35, 36
framework, 7, 8, 14, 18, 21, 22, 27, 29, 30, 35, 36
grammar, 6, 7, 36
hierarchy, 1, 8–10, 12, 14, 16, 18–21, 23, 25, 27, 29, 31, 34
meta-programming, 1, 6, 7, 16, 19, 23, 36
multi-method, 9
overloading, 12, 15, 32
polymorphism, 14, 16, 17
Spirit, 25, 36
Tiger, 7, 27, 30, 31, 33, 34
traversal, 1, 6–9, 12–14, 23, 24, 26–28, 30, 31, 34–36
Visitors, 7, 12, 14–16, 18, 21, 22, 25, 27, 29, 30, 32, 34–37