Generic Visitors in C++

Nicolas Tisserand <nicolas.tisserand@lrde.epita.fr>

LRDE seminar, May 28, 2003
## Table of Contents

**Introduction** ............................................................................................................. 4

**Visitor designs** .......................................................................................................... 5
  First attempt .................................................................................................................. 6
  Multi methods ................................................................................................................ 7
  The visitor design pattern ............................................................................................. 10
  Visitor combinators ...................................................................................................... 13
  What we want ................................................................................................................. 19

**Implementation techniques** ....................................................................................... 20
  Generic combinators ..................................................................................................... 21
• We have an abstract Syntax Trees (AST)

• We want to
  ▶ Walk the tree
  ▶ Perform actions on nodes (evaluation, printing ...)

→ We need an appropriate design
Visitor designs

- First attempt
- Multi methods
- Visitor design pattern
- Visitor combinators
First attempt

Naive design for a simple arithmetic expressions abstract syntax tree:

Adding a polymorphic operation requires modifying all the classes : (
Multi methods

- Generalized virtual methods (virtual on an arbitrary number of chosen parameters)

- Work like external methods added to classes

Imaginary example:

```cpp
void print (virtual IntExp*) { /* ... */ }
void print (virtual OpExp*) { /* ... */ }

Exp* e = new IntExp(51));
print (e); // dispatches to the first print method,
            // according to the dynamic type of e
```
Multi methods

- They are available in some languages (CLOS, Nice, Perl6) ...

- ... but not in C++ : ( C++ dispatches only the first (hidden) argument (this) of a method

- They can be emulated using various tricks (Alexandrescu, 2001)

→ We must find something else
Towards an object model

- Encapsulate each processing in one class
- Hierarchy of processings
- Separate AST and Processings
- We need to dispatch over two hierarchies

→ Someone may already have encountered this problem ...
The visitor design pattern

- Design patterns = Higher order abstractions
- Gamma et al. (Gamma et al., 1994)
- Also called Vanilla visitor
Visitor designs

The visitor design pattern

Design

\[
\text{Exp} \\
+\text{accept}(v:\text{ExpVisitor}): \\
\]

\[
\text{IntExp} \\
+\text{accept}(v:\text{ExpVisitor}): \\
\text{v.visit(this);} \\
\]

\[
\text{OpExp} \\
+\text{accept}(v:\text{ExpVisitor}): \\
\text{v.visit(this);} \\
\]

\[
\text{ExpVisitor} \\
+\text{visit}(\text{i:IntExp}): \\
+\text{visit}(\text{o:OpExp}): \\
\]

\[
\text{PrintVisitor} \\
+\text{visit}(\text{i:IntExp&}): \\
+\text{visit}(\text{o:OpExp&}): \\
\]

\[
\text{EvalVisitor} \\
+\text{visit}(\text{i:IntExp&}): \\
+\text{visit}(\text{o:OpExp&}): \\
\]
Discussion

• Benefits
  ▶ Double dispatch
  ▶ Decoupling of two inter-dependent hierarchies
  ▶ Factor out default traversal code by inheritance.

• Drawbacks
  ▶ Mixing of traversal and behavioral code
  ▶ No genericity
    * accept methods are bound to a specific abstract visitor class
    * visit methods are specific to the target hierarchy

→ We do not want to write traversal code each time we write a visitor
Visitor combinator

- Joost Visser, CWI (Visser, 2001) (→ StrategoXT)
- Break a monolithic visitor into small atomic visitors
- Use combinators to compose these visitors between them and get the final visitor
- A visitor combinator acts like a function from visitor to visitor
- Dummy example:
  
  \[
  \text{Compile} = \text{Sequence}(\text{Sequence}(\text{Escape, TypeCheck}), \text{Translate})
  \]
  
  \[
  \text{(Sequence : visitor * visitor -> visitor)}
  \]
struct Sequence : public ExpVisitor
{
    Sequence(ExpVisitor& first, ExpVisitor& second)
    : first_(first), second_(second) {}

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

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

    ExpVisitor& first_
    ExpVisitor& second_
};
Traversal combinators

• We now need to traverse nodes

• \textit{all}(v) applies \(v\) to all the children of the accepting node

• \textit{traversal} = \textit{all}(\textit{traversal})

• \textit{topdown}(v) = \textit{sequence}(v, \textit{all}(\textit{topdown}(v)))
  applies \(v\) to all the subtrees of the accepting node, in a top-down fashion

• \textit{bottomup}(v) = \textit{sequence}(\textit{all}(\textit{bottomup}(v), v)) applies \(v\) to all the subtrees of the accepting node, in a bottom-up fashion
Topdown example

prefix_print = toptdown(print)
             = sequence(print, all(prefix_print))

... once accepted by our example Exp AST

... outputs:
+  * 5 10 1
Conditional combinators

- `fail` throws a VisitFailure exception
- `sequence` works like “and”
- `choice(v1, v2)` let a node accept v1 firstly and, in case of failure, v2 secondly
- `choice` works like “or”
- Many other combinators exist: `one`, `try`...
Discussion

• Benefits
  ▶ Traversal and behavioral code cleanly separated
  ▶ Re-usability improved

• Drawbacks
  ▶ Too dynamic
  ▶ Hierarchy specific

→ We do not want to rewrite a visitor combinator framework for each target AST.
What we want

- Generic visitor combinators C++ library
- Instrument external visitable AST hierarchies
- **No** intrusion in target code
- Performance
- Ease of use

→ Use visitors combinators on any vanilla visitable hierarchy (like the LRDE Tiger compiler AST)
Implementation techniques

- Use genericity
- Adapt a foreign AST hierarchy
- Avoid dynamic binding
- Allow generic traversal
Generic combinators

- Some combinators (like identity or sequence) behave equally on any AST

- Wouldn’t it be possible to write them only once?

→ Let’s try to benefit from C++ static genericity
A generic Identity could be rewritten as follows:

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

However:

- This new combinator is still an abstract class
- `template virtual` methods are illegal
Writing generic combinators

- Generic combinators cannot be accepted directly by a target AST node
- Static and dynamic dispatch are not compatible
- We must implement an adapter between the two dispatches

→ We need a generic way to build concrete classes implementing any target visitor interface.
Acting as a Vanilla visitor

• Visitors of the target hierarchy can be described by:
  ▷ the abstract visitor type
  ▷ the list of visited types

• Meta C++ features
  ▷ parameterized inheritance
  ▷ static lists

→ Let’s use some meta-programming
Hierarchy Unrolling

We want a Visitor class parameterized by:

- **static list of types** `List<Type1, List<Type2, ...
     List<TypeN> ... )

- **the abstract visitor type** `AbstractVisitor`

... which generates, once instantiated:

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

template < typename L, typename V >
struct Visitor;

// Base specialization.
template < typename T, typename Tail, typename V >
struct Visitor < List < T, Tail >, V >
: public Visitor < Tail, V > {
    virtual void visit (T& t) { /* ... */ }
};

// Last element specialization.
template < typename T, typename V, typename H >
struct Visitor < List < T, Empty >, V > : public V {
    virtual void visit (T& t) { /* ... */ }
};
• The instantiated visitor inherits from N classes

• The virtual table is built once for the instantiated visitor

• No additional indirection

• Replace the “...” in the visit methods of the previous code by a delegation to an aggregated static visitor to make a static/dynamic adapter.

→ We can build arbitrary vanilla visitors without performance penalty
Static composition

- The type of parameters of many combinators (like sequence or choice) could be known at compile time

- We want to use this fact to avoid dynamic binding on visit methods calls

→ Let’s try to write a static choice combinator
Static composition

template < typename First, typename Second >
struct Choice {
    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_; 
};

Those combinators are then combined using expression templates
Self recursive combinators?

Static composition is nice, but ...

- self recursive types are **impossible** in C++

- we would like to write self recursive combinators

Think about: \(\text{traversal} = \text{all} \ (\text{traversal})\)

→ We need a way to “break” static composition
Self recursive combinator

- Concept coming from the Spirit parser framework (Spirit, 2001)

- Proxy for a static combinator

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

    AbstractCombinator* v_; // visit methods delegated to v_
};
```
Self recursive combinators

- We can now use self-recursion:

\[
\text{Combinator top\_down\_v} = v \&\& \text{all(top\_down\_v)};
\]

- \text{rhs type is Sequence < V, All < Combinator > >}

- a reference on \text{topdown\_v} can safely be taken before the end of its initialization

- An extra dynamic binding is used
Handy assignation

When trying to assign combinators, we can now write:

Combinator v = all(v1 && v2);

... instead of explicitly specifying the expression template return type:

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

Similar results could be possible with the non-standard typeof extension:

typeof(all(v1 && v2)) v = all(v1 && v2);
Traversal

- We want generic traversal combinators like all or one
- They need to know about node structures

→ We must find a way to describe the AST nodes
Adaptation

• Nodes can fall in three categories:
  ▶ n-ary nodes (number of children known at compile time)
  ▶ list nodes (dynamic number of children, iterator access)
  ▶ leaf nodes (no children)

• We describe them through traits specialization

• We use adapter classes, templated by static method pointers to ease traits specialization
Visitors

- Presentation
- Tiger use case
Presentation

- Target code (the AST to work on)
- Framework code (the *Visitors* library)
- Adapting code
- Client specific code
  - Custom generic combinators
  - Custom (AST specific) combinators
  - Visitor instantiations and use
Visitors Classes

![Diagram](image)

Generic Visitors in C++, Nicolas Tisserand - LRDE seminar, May 28, 2003 38
Tiger use case

Description of (a subset of) the Tiger AST:

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

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

```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);
```
**print_type visitor**

A combinator which prints the type of any node it visits:

```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;
```
**print_type visitor**

A combinator which prints the types of all the subtrees of the visited node:

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

... and its (demangled) output, when accepted by our sample Exp:

```cpp
ast::OpExp
ast::OpExp
ast::IntExp
ast::IntExp
ast::IntExp
```
A combinator which succeeds if the node is a constant expression and fails otherwise:

```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);
```
Problems

• There is a visit method for `decs_t (std::list-Decs *)` in the Visitor interface, but no accept method in `decs_t`

• Hybrid nodes like `FunctionDec` act simultaneously like a list node (the list of parameters) and like a n-ary node (two children: the result and the body)
Conclusion

- Applicability
- Future
Conclusion

Applicability

- Currently restricted to “well-formed” target AST hierarchies
- Writing adapting code for complex AST is harassing and error-prone
- Classical active libraries annoyances:
  - Slow compilation
  - Obfuscated code
  - Cryptic error messages
  - Compiler support
Future

Some possible improvements ...

- constness
- static concept checks
- node substitutions
- placeholders *a la* FC++: *(FC++, 2002)*

```cpp
Combinator top_down = sequence ( _1 , all (top_down (_1)))
```


References


Questions