Acyclic Visitor

(v1.0)

Robert C. Martin
Object Mentor
rmartin@oma.com

INTENT

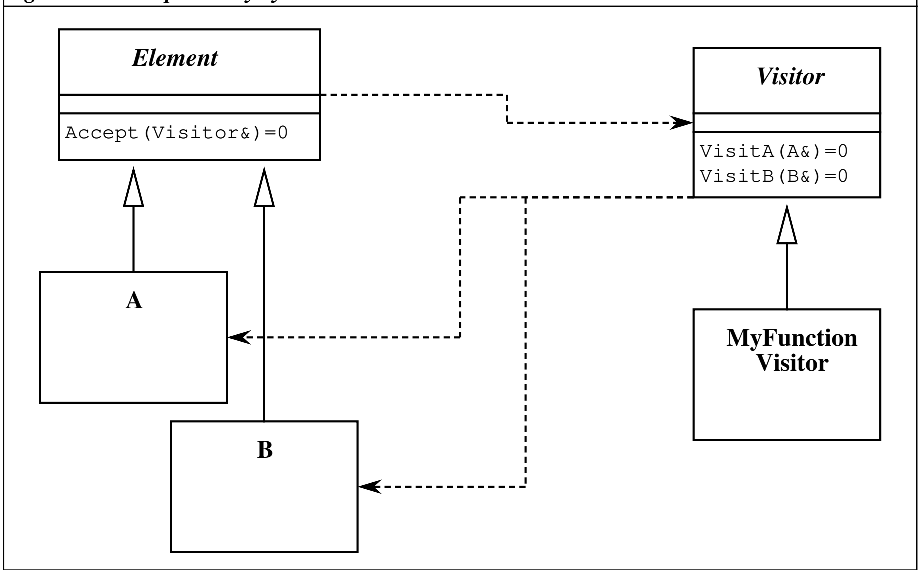
Allow new functions to be added to existing class hierarchies without affecting those hierarchies, and without creating the troublesome dependency cycles that are inherent to the GOF[1] VISITOR Pattern.

MOTIVATION

Procedural software can be written in such a way that new functions can be added to existing data structures without affecting those data structures. Object oriented software can be written such that new data structures can be used by existing functions without affecting those functions. In this regard they are the inverse of each other. Adding new data types without affecting existing functions is at the heart of many of the benefits of OO. Yet there are times when we really want to add a new function to an existing set of classes without changing those classes. The VISITOR[2] pattern provides a means to accomplish this goal.

However, the VISITOR pattern, when used in static languages like C++, Java, or Eiffel, causes a cycle in the source code dependency structure. (See Figure 1 and the legend at the end of this paper.) A source code dependency means that the source code of one module must refer to (via `#include`, or `import`, or some other mechanism) the source code of another module.

Figure 1: The dependency cycle in the VISITOR ***Pattern***



1. "Gang of Four" : Gamma, Helm, Vlissides, and Johnson. The four authors of *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1995
2. *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1995 p. 331

The dependency cycle in this case is as follows:

- The base class of the visited hierarchy (`Element`) depends upon the base class of the corresponding visitor hierarchy (`Visitor`).
- The `Visitor` base class has member functions for each of the derivatives of the `Element` base class. Thus the Visitor class depends upon each derivative of `Element`.
- Of course, each derivative of `Element` depends upon `Element`.

Thus we have a cycle of dependencies that causes `Element` to transitively depend upon all its derivatives.

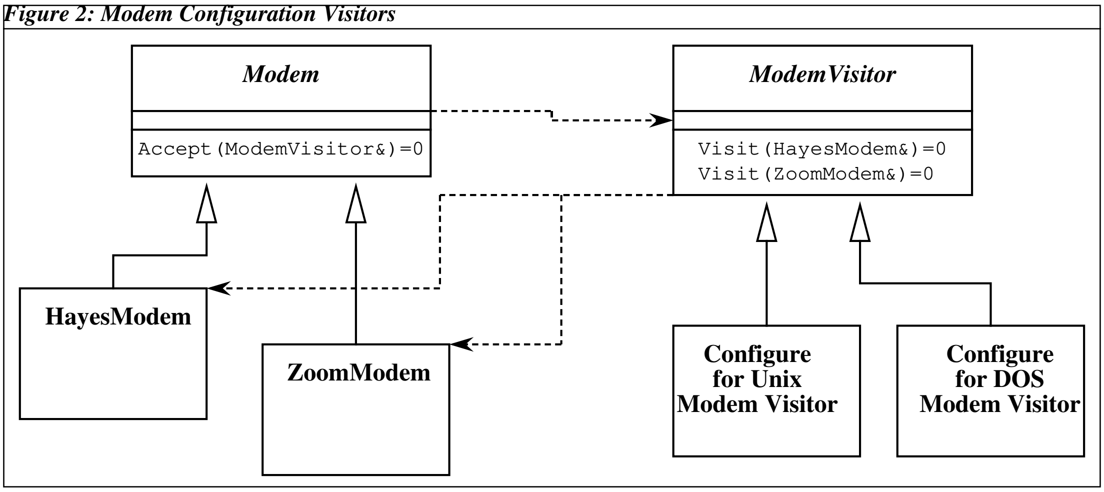
This knot of dependencies can cause significant troubles for the programmer who must maintain the code which contains them. Any time a new derivative of `Element` is created, the `Visitor` class must also be changed. Since `Element` depends upon `Visitor`, every module that depends upon `Element` must be recompiled. This means that every derivative of `Element`, and possibly every *user* of every derivative of `Element,` must also be recompiled.

Where possible, this dependency cycle should be mitigated by using *forward declarations*. That is, in many cases the `Element` base class can *forward declare* the `Visitor` base class, and the `Visitor` base class can *forward declare* the derivatives of `Element`. This creates a much weaker source code dependency that Lakos[3] refers to as a *name only* dependency. Although weaker, this is still a dependency cycle and still causes many of the problems mentioned in the last paragraph. Specifically, even when *name only* dependencies are used as much as possible, every time a new derivative of `Element` is created, all the existing derivatives of `Element` must be recompiled[4].

Partial Visitation

Another disadvantage to the dependency cycle created by the Visitor pattern is the need to address *every* derivative of Element in *every* derivative of Visitor. Often, there are hierarchies for which visitation is only required for certain derivatives of Element. For example, consider a modem hierarchy (See Figure 2).

Figure 2: Modem Configuration Visitors



Here we see a very compelling use for VISITOR. We have a typical hierarchy of `Modem` classes with one derivative for each modem manufacturer. We also see a hierarchy of visitors for the `Modem` hierarchy. In this example, there is one visitor that adds the ability to configure a modem for Unix; and another that adds the ability to configure a modem for DOS. Clearly, we do not want to add these functions directly to the `Modem` hierarchy. There is no end to such functions! The last thing we want is for every user of `Modem` to be recompiled every time a new operating system is released. Indeed, we don't want `Modem` to know anything at all about operating systems. Thus, we use VISITOR to add the configuration function to the `Modem` hierarchy without affecting that hierarchy.

However, VISITOR forces us to write a function for the cross product of all `Modem` derivatives and all `ModemVisitor` derivatives. i.e. we need to write the functions that configure every type of modem to every type of operating

3. *Large Scale C++ Software Design*, John Lakos, Addison Wesley, 1996. p249.
4. See: "What's wrong with recompiling?", in the Notes section.

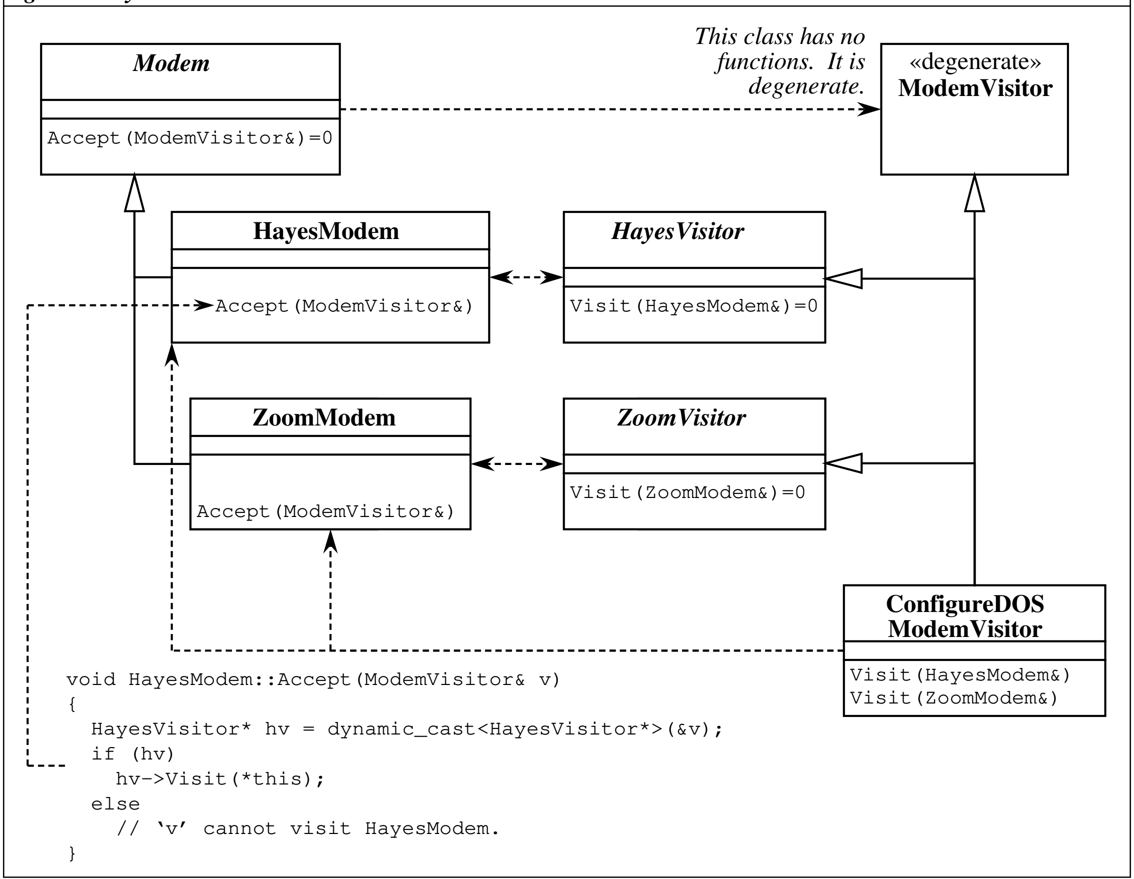
system. However, what if we never use Hayes modems with Unix? VISITOR will still force us to write a function to do it! We could, of course, print an error from the function in the Visitor base class, and then allow that function to be inherited, but we still have to write that function.

Now consider a much larger hierarchy, one in which the cross product of `Element` derivatives and `Visitor` derivatives is sparsely populated. The VISITOR pattern may become inconvenient in such a hierarchy because every visitor depends upon every derivative of `Element`. Any time a new derivative of `Element` is added *all* derivatives, even derivatives which do not require visitor functions, must be recompiled. We would prefer to write only the functions that need writing and keep them independent from all the other derivatives of `Element`.

Solution

These problems can be solved by using multiple inheritance and `dynamic_cast`. (See Figure 3)

Figure 3: Acyclic Modem Visitor



Here we see how the dependency cycle can be broken. Rather than put pure virtual functions into the `ModemVisitor` class, we make it completely degenerate; i.e. it has no member functions at all! We also create one abstract class for each derivative of `Modem`. These classes, `HayesVisitor` and `ZoomVisitor`, provide a pure virtual `Visit` function for `HayesModem` and `ZoomModem` respectively. Finally we inherit all three of these classes into the `ConfigureDOSModemVisitor`. Note that this class has exactly the same functions that it had in Figure 2. Moreover, they are implemented in exactly the same way.

The `Accept` function in the derivatives of `Modem` use `dynamic_cast` to cast *across* the visitor hierarchy from `ModemVisitor` to the appropriate abstract visitor class. Note: this is not a downcast - it is a *cross* cast. It is one of the great benefits of dynamic_cast that it can safely cast to any class anywhere in the inheritance structure of the object it operates on.

Now what happens if we never use Hayes modems with Unix? The `ConfigureUnixModemVisitor` class will simply not inherit from `HayesVisitor`. Any attempt to use a Hayes modem with Unix will cause the `dynamic_cast` in `HayesModem::Accept` function to fail, thus detecting the error at that point.

There are no dependency cycles anywhere in this structure. New `Modem` derivatives have no affect on existing modem visitors unless those visitors must implement their functions for those derivatives. New `Modem` derivatives can be added at any time without affecting the users of `Modem`, the derivatives of `Modem`, or the users of the derivatives of `Modem`. The need for massive recompilation is completely eliminated.

APPLICABILITY

This pattern can be used anywhere the VISITOR pattern can be used:

- When you need to add a new function to an existing hierarchy without the need to alter or affect that hierarchy.
- When there are functions that operate upon a hierarchy, but which do not belong in the hierarchy itself. e.g. the ConfigureForDOS / ConfigureForUnix / ConfigureForX issue.
- When you need to perform very different operations on an object depending upon its type.

This pattern should be preferred over VISITOR under the following circumstances:

- When the visited class hierarchy will be frequently extended with new derivatives of the Element class.
- When the recompilation, relinking, retesting or redistribution of the derivatives of `Element` is very expensive.

STRUCTURE (See Figure 4.)

PARTICIPANTS

- **Element**. The base class of the hierarchy which needs to be visited. Visitors will operate upon the classes within this structure. If you are using visitor to add functions to a hierarchy, this is the base class of that hierarchy
- **E1, E2, ...** The concrete derivatives of `Element` that require visiting. If you are using visitor to add functions to a hierarchy, you will write one function for each of these concrete derivatives.
- **Visitor**. A degenerate base class. This class has no member functions at all. It sole purpose is as a place holder in the type structure. It is the type of the argument that is taken by the `Accept` method of `Element`. Since the derivatives of `Element` use this argument in a `dynamic_cast` expression, `Visitor` must have at least one virtual function -- typically the destructor.
- **E1Visitor, E2Visitor, ...** The abstract visitors that correspond to each of the concrete derivatives of `Element`. There is a one to one relationship between these classes. Each concrete derivative of `Element` will have a corresponding abstract visitor. The abstract visitor class will have one pure virtual `Visit` method that takes a reference to the concrete `Element` derivative.
- **VisitForF.** This is the actual visitor class. It derives from `Visitor` so that it can be passed to the `Accept` function of `Element`. It also derives from each of the abstract visitors that correspond to the concrete classes that this visitor will visit. There is no need for the visitor to derive from all the abstract visitor classes; it only needs to derive from the ones for which it will implement `Visit` functions.

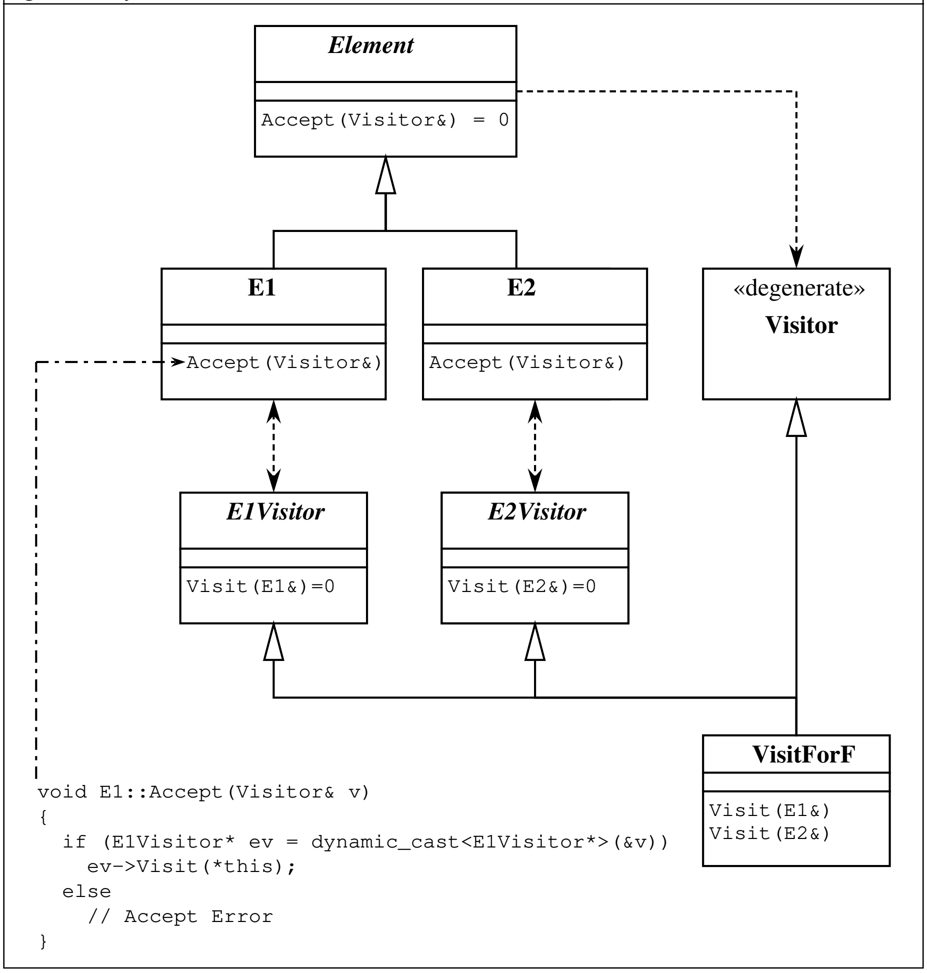
COLLABORATIONS

1. The process begins when a user wishes to apply one of the visitors to an object in the `Element` hierarchy. The user does not know which of the concrete derivatives of `Element` it actually has; instead is simply has a reference (or a pointer) to an `Element`.

2. The user creates the visitor object. (e.g. `VisitForF` in Figure 4) The visitor object represents the function that the user would like to invoke upon the `Element`.

3. The user sends the `Accept` message to the `Element` and passes the visitor object as a reference to a `Visitor`.

Figure 4: Acyclic Modem Visitor



4. The `Accept` method of the concrete derivative of `Element` uses `dynamic_cast` to cast the `Visitor` object to the appropriate abstract visitor class (e.g. `E1Visitor` from Figure 4).

5. If the `dynamic_cast` succeeds, then the `Visit` message is sent to the visitor object using the interface of the abstract visitor class. The concrete derivative of `Element` is passed along with the `Visit` message.

6. The actual visitor object executes the `Visit` method.

CONSEQUENCES

The consequences of this pattern are the same as those for VISITOR with the following additions:

+ All dependency cycles are eliminated. Derivatives of Element do not depend upon each other. Recompilation is minimized.
+ Partial visitation is natural and does not require additional code or overhead.
– `dynamic_cast` can be expensive in terms of runtime efficiency. Moreover, its efficiency may vary as the class hierarchy changes. Thus, ACYCLIC VISITOR may be inappropriate in very tight real time applications where run time performance must be predictable
– Some compilers don't support dynamic_cast.
– Some languages don't support dynamic type resolution, and/or multiple inheritance.
– In C++, the `Visitor` class must have at least one virtual function. Since the class is also degenerate, we typically make the destructor virtual.

– Use of this pattern implies that there will be an abstract visitor class for each derivative of Element. Thus, classes tend to proliferate rapidly.

Sample Code

The following is the code for the Modem example used in Figure 3.

```
// Visitor is a degenerate base class for all visitors.
class Visitor
{
  public:
    virtual ~Visitor() = 0;
    // The destructor is virtual, as all destructors ought to be.
    // it is also pure to prevent anyone from creating an
    // instance of Visitor.  Since this class is going to be
    // used in a dynamic_cast expression, it must have at least
    // one virtual function.
};

class Modem
{
  public:
    virtual void Accept(Visitor&) const = 0;
};

class HayesModem;
class HayesModemVisitor
{
  public:
    virtual void Visit(HayesModem&) const = 0;
};

class HayesModem : public Modem
{
  public:
    virtual void Accept(Visitor& v) const;
};

void HayesModem::Accept(Visitor& v) const
{
  if (HayesModemVisitor* hv = dynamic_cast<HayesModemVisitor*>(&v))
    hv->Visit(*this);
  else
    // AcceptError
}

class ZoomModem;
class ZoomModemVisitor
{
  public:
    virtual void Visit(ZoomModem&) const = 0;
};

class ZoomModem : public Modem
{
  public:
    virtual void Accept(Visitor& v) const;
};

void ZoomModem::Accept(Visitor& v) const
{
  if (ZoomModemVisitor* zv = dynamic_cast<ZoomModemVisitor*>(&v))
```

```
    zv->Visit(*this);
  else
    // AcceptError
}

//------------------------
// ConfigureForDOSVisitor
//
// This visitor configures both Hayes and Zoom modems
// for DOS.
//
class ConfigureForDosVisitor : public Visitor
                             , public HayesModemVisitor
                             , public ZoomModemVisitor
{
  public:
    virtual void Visit(HayesModem&); // configure Hayes for DOS
    virtual void Visit(ZoomModem&);  // configure Zoom for DOS
};

//-------------------------
// ConfigureForUnixVisitor
//
// This visitor configures only Zoom modems for Unix
//

class ConfigureForUnixVisitor : public Visitor
                              , public ZoomModemVisitor
{
  public:
    virtual void Visit(ZoomModem&); // configure Zoom for Unix
};
```

KNOWN USES

We have used this pattern in several of the projects we have consulted for. It has been used in the design of the "Mark Facility Controller" created by the Toolkit Working Group at Xerox. It has also been used in the ETS/NCARB project.[5]

NOTES

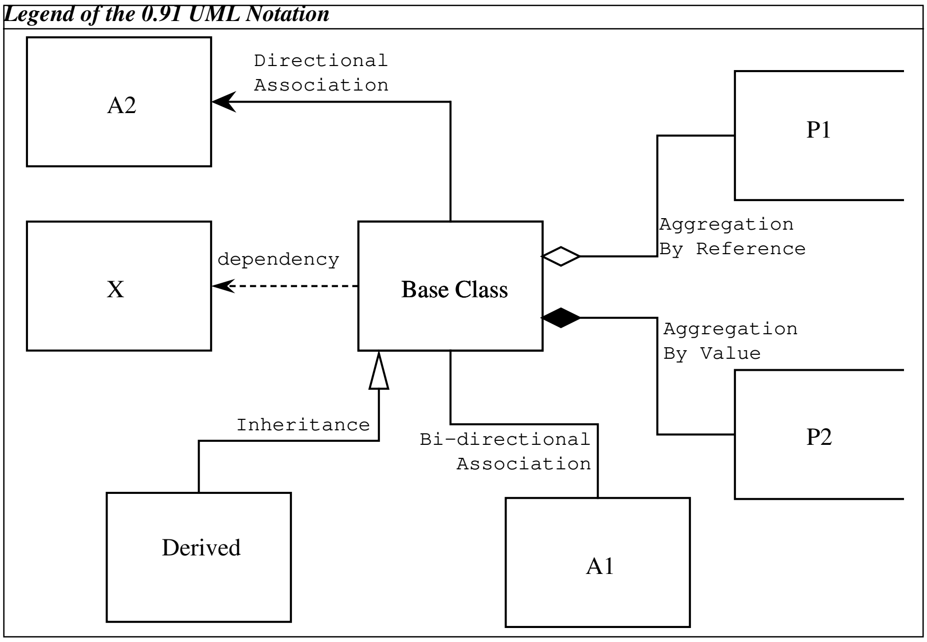
This pattern solves a particularly nasty problem of tangled dependencies. I find this interesting in light of the fact that it depends on two such controversial features. The pattern would not be possible were it not for *multiple inheritance* and *run time type information*; both of which have been attacked as being "non-OO".

What's wrong with recompiling?

Recompiles can be very expensive for a number of reasons. First of all, they take time. When recompiles take too much time, developers begin to take shortcuts. They may hack a change in the "wrong" place, rather than engineer a change in the "right" place; simply because the "right" place will force a huge recompilation. Secondly, a recompilation means a new object module. In this day and age of dynamically linked libraries and incremental loaders, generating more object modules than necessary can be a significant disadvantage. The more DLLs that are affected by a change, the greater the problem of distributing and managing the change. Finally, a recompile means a new release of every module which needed recompiling. New releases require documentation, and testing; causing potentially huge amounts of manpower to be invested.

5. see 'publications' at www.oma.com

LEGEND



Legend of the 0.91 UML Notation
A2
Directional
Association
P1
X
dependency
Base Class
Aggregation
By Reference
Aggregation
By Value
Inheritance
Bi-directional
Association
P2
Derived
A1