

The Visitor pattern

The Visitor pattern is an architectural technique enabling you to define arbitrary facilities applicable to instances of existing classes.

The idea is very simple: **let the operations know about the types**. If we are to apply various operations to various types, either each operation must know about every applicable type, or the other way around. With dynamic binding, each *type* knows about the applicable operations. Now we are concerned with the other case: a client class needs the ability to perform an operation on instances of many possible classes (the target classes); we may call such objects — such as taxis and trams in our staple example — *target objects*, or just *targets*.

The issue is not how to define the operations. We have to assume that appropriate algorithms are available to write features such as

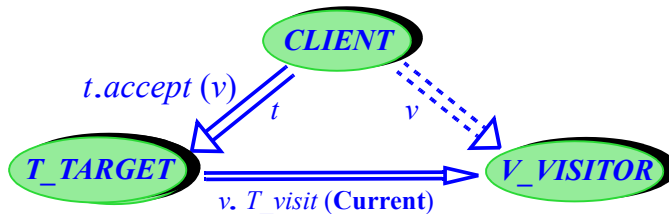
```
flash_taxi (t: TAXI) do ... Algorithm for flashing a taxi ...end
```



[8]

and so on (*flash_tram, flash_bus...*). The question is about **architecture**: where do the features belong, and how can we use them in a way that preserves the extendibility of the software?

The features are not in the target classes: if they were, dynamic binding would be the solution. We are assuming this is not the case; that is why they must get their targets through arguments, such as *t: TAXI* above. But there is no reason to require *client* classes to implement the features either: better collect the features in a separate class which knows how to perform a specific operation, say flash, on various possible targets such as taxis and trams.

So our *pas-de-deux* between the client and the target turns into a *ménage-à-trois* between client, target and **visitor**. A visitor is an object able to apply a *single* kind of operation to *many* kinds of object; this is the reverse of classes designed for dynamic binding, although — as testimony to the power of the basic O-O ideas — visitors are still implemented as classes and the scheme still crucially relies on dynamic binding.



 Client (calls)
 Client (knows about)

**The Visitor
ménage-à-trois**

(→ See page 611 for the full picture with inheritance.)

→ For more about the concept of pattern see “About design patterns”, page 678.

For a target type *T*, such as *TAXI*, and an operation *V*, such as *flash*, the figure shows the interplay between:

- The target class, *T_TARGET*, representing target objects of type *T*. Class *TAXI* is a typical example.
- The visitor class *V_VISITOR*, for example *FLASH_VISITOR*, representing application of the chosen operation to objects of many different types.
- The client class, representing an application element that needs to perform the operation on target objects of various types.

Often the client class will need to perform the operation on a *set* of target objects, for example flash all Traffic objects in a list. This explains the term *visitor*: a visitor object — an instance of a class such as *FLASH_VISITOR* — enables the client to “visit” every element of a certain structure, each time performing the appropriate version of a specified operation. As you know the process of performing such visits is called “iteration” or “traversal”.

← “Definition: Iterating”, page 397; “Traversals”, page 453.

As always in discussing software architecture for extendibility and reusability, it is important to examine who knows what, and also who does **not** need to know what, with the aim of reducing the amount of knowledge that is spread over the structure and would cause trouble when the information changes. Here:

- The target class *knows* about a specific type, such as *TAXI*, and also (since for example *TAXI* inherits from *VEHICLE* and *VEHICLE* from *MOVING*) its context in a type hierarchy. It *does not know* about new operations requested from the outside, such as flashing.
- The visitor class *knows* all about a given operation, and provides the appropriate variants for a range of relevant types, denoting the corresponding objects through arguments: this is where we will find routines such as *flash_bus*, *flash_tram*, *flash_taxi*. It *does not know* anything about clients.
- The client class needs to apply a given operation to objects of specified types, so it must *know* these types (only their existence, *not* their other properties) and the operation (only its existence and applicability to the given types, *not* the specific algorithms in each case).

Using the Visitor pattern, the client will be able to apply the operation, for example, to all items of applicable types in a list, without knowing these types individually, as in

```

flash_all (fl: LIST [ TARGET ] -- See below about TARGET
-- Flash all items in fl.
do
  from fl.start until fl.after loop
    -- "Flash fl.item"
    fl.forth
  end
end

```

← The line partially in red is pseudocode, see "Definition: Pseudocode", page 108. The expansion of the pseudocode comes next.

The Visitor pattern provides an implementation of the line in pseudocode. That implementation is very simple (follow it in the figure): the basic visit operation is

```
t.accept (v)
```

for a target object *t* and a visitor object *v*, leading us to replace the pseudocode line by

```
fl.item.accept (flasher)
```

where *flasher* is the *FLASH_VISITOR* object.

All target classes must provide a feature *accept*, whose general implementation is

```

accept (v: VISITOR)
-- Apply the relevant visit operation from v to x.
do
  v. T_visit (Current)
end

```

T_visit is a visitor feature that implements the requested operation for the type *T*: for example *bus_visit*, *tram_visit*. These procedures must be provided on the visitor side:

```

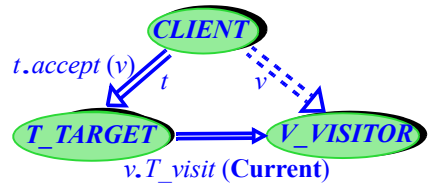
bus_visit (t: BUS ) do flash_bus (t) end
tram_visit (t: TRAM ) do flash_tram (t) end
taxi_visit (t: TAXI ) do flash_taxi (t) end
... and so on ...

```

It is generally possible to avoid wrapping existing routines in this way, and instead directly implement *flash_bus* under the name *bus_visit* etc.

Admire the delicately choreographed duet in which the target and the visitor engage once the client has set them in motion. The target object knows about its own type; it does not know the requested operation, but knows someone who

Visitor classes



(Figure from page 608.)

knows: the visitor v passed by the client as the argument to *accept*. So it calls T_visit on the visitor, where T identifies the target type, and passes itself — **Current** — as argument. This enables the visitor to use the right operation, identified by the inclusion of T in the routine name, to the right object, identified by the argument to that routine.

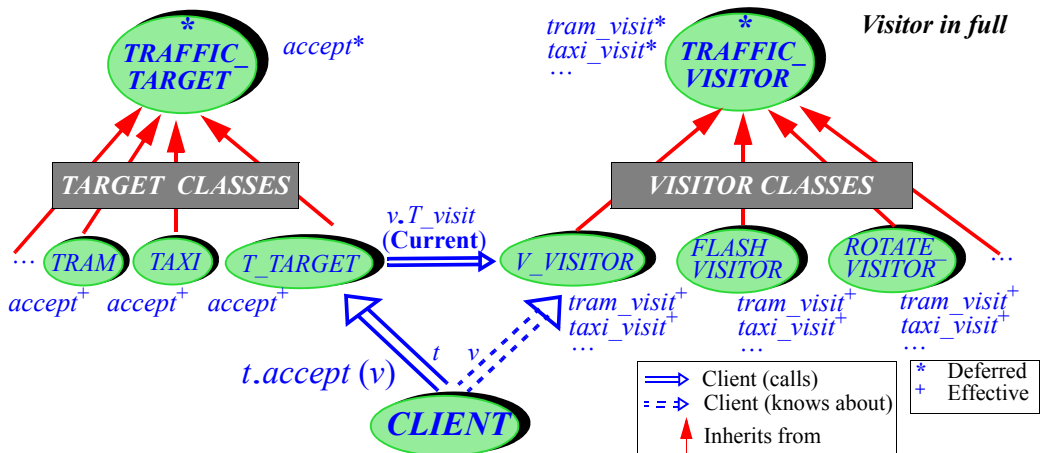
Even though the Visitor pattern is intended to remedy limitations of dynamic binding, it fundamentally relies on dynamic binding. Twice, in fact:

D1 In the client's call $t.accept(v)$, to select the right target.

D2 In the target's call $v.T_visit(\text{Current})$, to select the right operation (by selecting the right visitor).

Dynamic binding is also known as dynamic *dispatch*, and more specifically as **single dispatch** since it dispatches a call to the suitable algorithm on the basis of a single criterion (the type of the call's target). The Visitor pattern is an example of **double dispatch**: selecting an action on the basis of *two* criteria, here a kind of object and a kind of operation. It illustrates a possible technique for achieving double dispatch in a framework that supports single dispatch, as most object-oriented languages do: use single dispatch *twice*. The first call, **D1**, taking as its *target* the target object t , performs a dispatch on the first criterion by applying dynamic binding to that object; it includes the other operand, the visitor v , as its *argument*. This allows the routine, here *accept*, to perform the second dispatch through a second call, **D2**, which uses this argument as its own target. The routine T_visit of that final call must still have access to the original target object; this is achieved by passing **Current** as the argument.

For the first case of dynamic binding, **D1**, to work, all target classes must have a feature *accept*, each redeclaring it in the form $v.T_visit(\text{Current})$ as above.



As illustrated by the figure, *accept* will come from a common ancestor, where it is deferred. That ancestor is class *TARGET*, which can be very simple:

Appearing as
TRAFFIC_TARGET;
see explanation below.

```

note
  description: "Objects that can be used as targets in the Visitor pattern"
deferred class TARGET feature
  accept (v: VISITOR )
    -- Make v perform one "visit" operation on the current object
    --| Note: typical implementation is v.T_visit (Current)
    --| where T is the specific effective descendant type.
  deferred
  end
end

```

The last two lines of the header comment rely on a standard convention: starting with *--|* rather than just *--* will cause the contract view not to display them. This is appropriate for comments that describe properties of the implementation, not relevant for clients.

← "What characterizes a metro line",
page 53.

Having to make all target classes inherit from a special *TARGET* class is disappointing, since the original idea was to reuse target classes as they are. With the technique as seen so far, if you have no say at all on target classes, you are stuck. This is the principal limitation of the Visitor pattern; to remove it, we will need to go to completely different techniques, as previewed below. In many practical cases, though, it is not as bad as it sounds: the aim is to avoid modifying target classes again and again, every time a new kind of visitation need pops up. Here you must only make sure that the target classes have **one** ancestor with one specific feature; then for the rest of their lives you can add visitors to you heart's content.

On the visitor side too you need a common ancestor, say *VISITOR*, and necessary to make *accept* valid in *TARGET*. Here it is not a problem to require such a common ancestor, since you will need to write specific visitor classes for every case of applying the Visitor pattern.

TRAFFIC_VISITOR
in the figure, see next.

The figure on the previous page extends our earlier one by including all relevant classes and inheritance relationships. The names of the deferred classes *TARGET* and *VISITOR* now start with *TRAFFIC_*, to be replaced by any name identifying the application when you use the Visitor pattern; this is because with the techniques seen so far it is really impossible to define these classes as reusable components with full generality. *VISITOR* in particular must know all the target types in the application so that it can list, even in deferred form, the relevant features, here *tram_visit*, *taxi_visit* and such.

As before, *T* and *V* stand for prototypical examples of a target type and an operation, complemented here by concrete examples such as *TRAM* and *FLASH*.

This completes the presentation of the Visitor pattern; I hope that you understand it thoroughly, not just the technique but its precise goals, scope, principles, advantages and limitations, as well as how to apply it in practice.

Improving on Visitor

The Visitor design pattern is a popular and useful technique, but suffers from the two limitations noted:

- To ensure visitability, all the target classes must descend from a common ancestor. This is not realistic if they are owned by someone else who has not prepared them accordingly.
- The pattern is not a *reusable* solution: it must be programmed anew for each use, with code that is very similar-looking in all cases; in particular, all implementations of *accept* in target classes follow the same scheme.

It is possible to improve on this basic scheme by using genericity. Beyond this, however, a full satisfactory solution relies on the mechanism studied in the next chapter: **agents**.

→ “Generic visitor”,
16-E.8, page 618

The agent-based solution is easy to use: for each applicable target type *T*, write a *visit* routine that implements the requested operation for *T*. This does not require modifying any existing class or adding any class. Then, to apply the operation variants to many different targets, pass both the target and **agent visit**, an object representing the operation, to a general mechanism that applies an agent to an object without having to know anything specific about either. An exercise in the agent chapter asks you to pursue this solution further. The “pattern library” developed at ETH provides a reusable visitor solution implementing this approach.

→ “Visiting with
agents”, 17-E.7,
page 660.

16.15 FURTHER READING

Bertrand Meyer: *Object-Oriented Software Construction* (Second edition, Prentice Hall, 1997).

Contains a detailed analysis of inheritance over several chapters.

Bertrand Meyer and Karine Arnout: *Componentization: the Visitor Example*, in *Computer* (IEEE), vol. 39, no. 7, July 2006, pages 23-30. Also available at se.ethz.ch/~meyer/publications/computer/visitor.pdf.

The Visitor pattern complements dynamic binding by making it easy to add an operation to a set of existing types (rather than the reverse). This article presents the pattern and proposes a reusable component that provides it, part of the ETH “pattern library”. You will find many other descriptions of the Visitor pattern in the literature (including in Wikipedia); most of them derive their descriptions from E. Gamma et al., *Design Patterns*, Addison-Wesley, 1994.