Department of Computer Science
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

COS121 Lecture Notes: L31

Visitor Design Pattern

24 October 2014

Contents

L31.1 Introduction

In this lecture you will learn about the Visitor Design Pattern. This pattern separates the bahaviour of the elements in an aggragate from the state of these elements. This is done with the intention to simplify the maintainance when the behaviour of these elements have to be changed or extended. This is done with extreme elegance. When implemented correctly neither the elements of the aggregate nor the clients using the aggregate need to be recompiled when new functions are added to the system. Unfortunately the application of this pattern complicates the maintainance of the aggragate itself. It is difficult to add classes to the aggragte. Thus, this pattern is more applicable in a system with changing processing needs and a stable internal structure of elements. I.e. a system where you will seldom add new classes but have the need to often add new functions to some derived classed in an aggregate and consequently new virtual functions to exsiting interfaces to the aggregates.

L31.2 Programming Preliminaries

L31.2.1 Single dispatch

With the exception of the Vistor design pattern, all the design patterns that uses delegateion as their strategy, the concrete function that is called from a function call in the code depends on the dynamic type of a single object and therefore they are known as single dispatch calls, or simply virtual function calls.

Single dispatch is a natural result of function overloading. Function overloading allows the function called to depend on the type of the argument. Function overloading however is done at compile time where the compiler creates code for each overloaded version of the function. Consequently there is no runtime overhead because there is no name collision. Calling an overloaded function goes through at most one virtual table just like any other function.

L31.2.2 Double disapatch

In the Visitor design pattern the mechanism that dispatches a function call to different concrete functions depends on the runtime types of two objects that are involved in the call. This mechanism is known as double dispatch.

When double dispatch is applied two overloaded functions are involved in the process to execute the correct concrete function. Assume there are two class hierachies respectively with abstract classes called `HierarchyA` and `HierarchyB`. Further assume that `functionA(:HierarcyB)` and `functionB(:HierarchyA)` are virtual functions respectively defined in `HierarchyA` and `HierarchyB`. A double dispatch call to a concrete class derived from `HierarchyB` involves calling `functionB(:HierarchyA)` and passing a pointer to a concrete class dervied from `HierarchyA` via the parameter. This will result in the execution of the correct concrete implementation of `functionB(:HierchyA)`. The next step in the double dispatch process will now use the parameter that was passed to `functionB(:HiercahyA)` to call back. i.e. if `objectA : HierrachyA` is the argument

that was passed to `functionB(:HierchyA)`, the body of `functionB(:HierchyA)` will include a a statement like `objectA->functionA(objectB)`. This call will in turn use the type of its argument to determine the correct concrete implementation of `functionA(:HierarchyB)` to be executed.

L31.3 Visitor Design Pattern

L31.3.1 Identification

Name	Classification	Strategy
Visitor	Behavioural	Delegation
Intent		
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. ([2]:331)		

L31.3.2 Structure

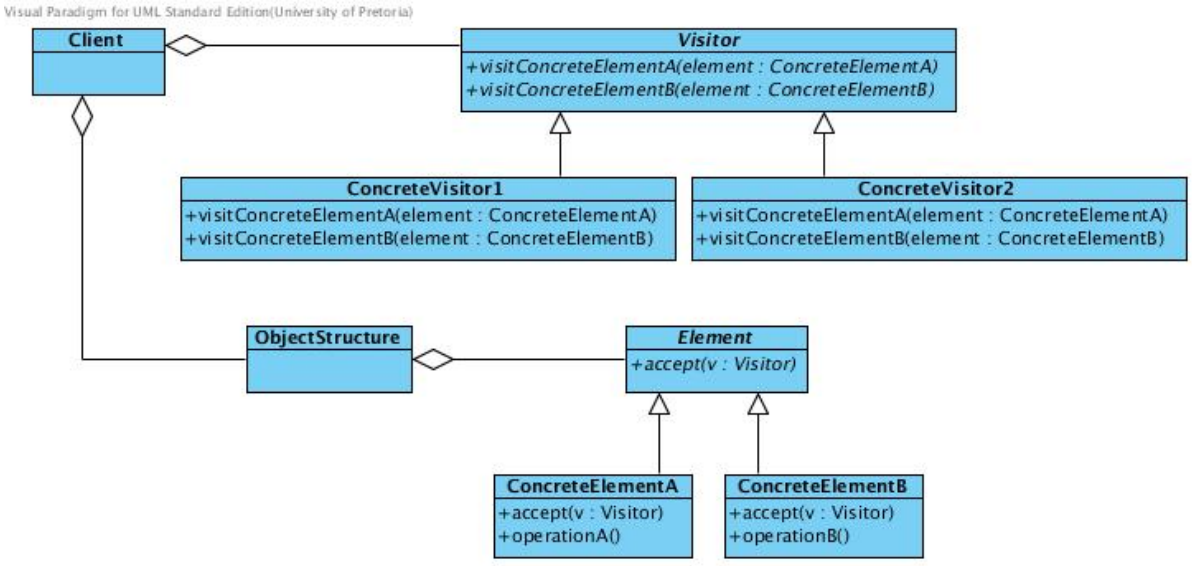

Figure 1: The structure of the Visitor Design Pattern

L31.3.3 Problem

Many distinct and unrelated operations need to be performed on node objects in a aggregate structure that may be heterogenious. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation [3].

L31.3.4 Participants

Visitor

- Each class of ConcreteElement has a `visit()` operation declared for it.
- The operation's signature identifies the class that sends the `visit()` request to the visitor.
- The particular class is then accessed through the interface defined for it.

ConcreteVisitor

- Implements the operations defined by visitor.
- May store information about objects that are visited.

Element

- Defines an `accept()` operation that takes an object of Visitor as a parameter.

ConcreteElement

- Implements the `accept()` operation that takes an object of Visitor as a parameter.

ObjectStructure

- Has a highlevel interface that allows the Visitor access and traverse its elements.
- This structure may be a Composite or a collection such as an array, list or a set.

L31.4 Visitor Pattern Explained

L31.4.1 Improvements achieved

- The operations of a conceptual operation is kept together rather than being scattered in different clases in an aggregate. Thus cohesion is increased because related operations are logically grouped in different visitors.
- The different players are independent. This independence reduces coupling [1]
- Aside from potentially improving separation of concerns, the visitor pattern has an additional advantage over simply calling a polymorphic method: a visitor object can have state. This is extremely useful in many cases where the action performed on the object depends on previous such actions [5].

L31.4.2 Disadvantages

- It is difficult to change the aggregate. When classes in the aggregate are changed *all* vistors need to be changed. This is the case even when the changes are part of the aggregate which is of no particular interest to the visitor [4].
- The encapsulation of the concrete elements is diminished to allow the visitors to perform their operations [2].
- Owing to double dispatch (i.e. twofold redirection), the application of the Vistor pattern may introduce a significant performance penalty. One might say that this is the price of complete flexibility (which may or may not be worth paying).

L31.4.3 Implementation Issues

The Object stucture is dependant on the Visitor to compile as it has visitors as parameters to its `accept()` methods. The Visitor in turn is also dependant on the Concrete Elements in the Object structure to compile for the same reason. This is called a *cyclic dependancy*. Fortunately they depend only on the names of the classes. Therefore, the problem can be avoided by including a forward declaration of the Visitor class in the Object structure and first compiling the Object structure or vice versa.

Note that it is required that the Visitor participant defines a separate `visit()` function for each of the concrete element types in the aggregate. This function also takes a parameter of the type of this concrete element. Owing to C++ overloading, these functions may all have the same name and can be distiguished by their differing parameter types. It is, however, important to note that the `accept()` methods in the elements still need to be implemented in the concrete classes despite the fact that they seem to be identical. If you would implement it in the interface, the static type `*this` would be the base class which would not provide the compiler with the needed type information.

L31.4.4 Related Patterns

Composite
: The Coposite is supportive to the Visitor. Visitors can apply an operation over an object structure defined by the Composite pattern.

Iterator
: Iterator and Visitor has similar intents. Visitor, however, is more general than Iterator. 'n Iterator is restricted to operations on elements of the same kind while Visitor can operate on elements of different types.

Interpreter
: The Visitor pattern may be applied to do the interpretation.

Abstract Factory
: Abstract Factory and Visitor has similar structure. Abstract factory applies the structure to create families of objects while Vistor applies this structure to perform a group of related operations.

Bridge

Both Bridge and Visitor separates state and behaviour of objects. Bridge applies single dispatch while Vistor applies double dispatch.

L31.5 Example

Figure 2: Class Diagram of a system illustrating the implementation of the Visitor design pattern

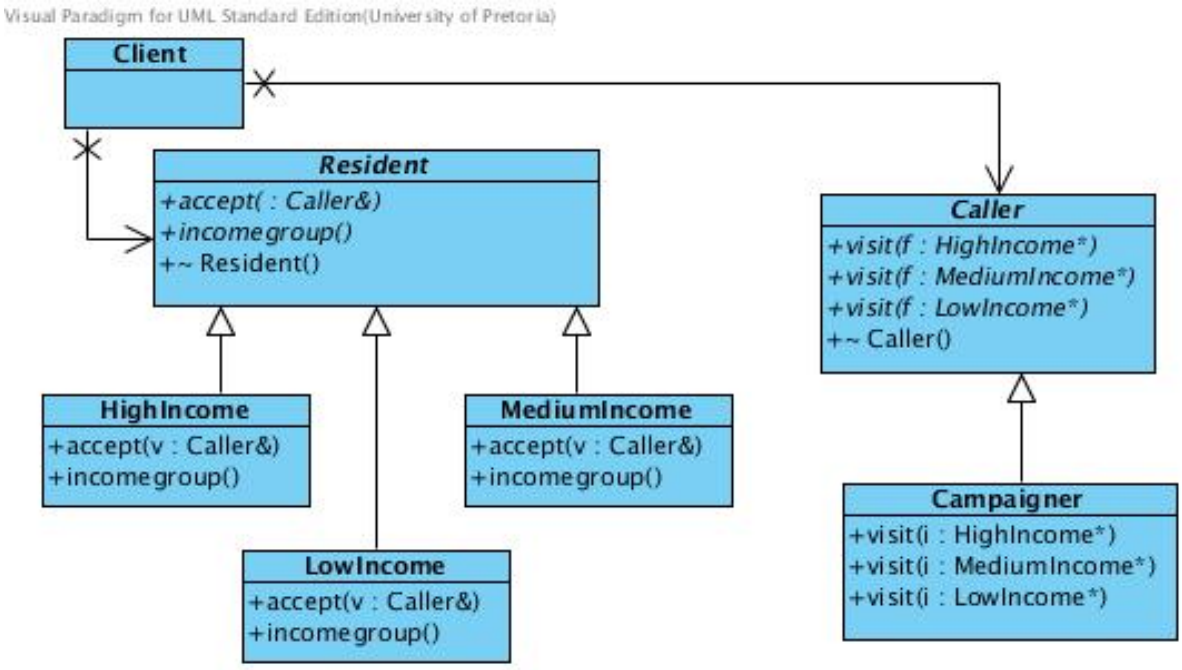

Figure 2 is a class diagram of an application that implements the visitor design pattern. It illustrates an implementation of the Visitor design pattern by showing that a Campaigner might have different actions to take while visiting different kinds of voters. In this system it will **not** be easy to change the structure of the Resident class hierarchy but it will be very easy to change the behaviour of the Campaigner or add a different kind of visitor without having to change any code in the Resident class hierarchy.

Participant	**Entity in application**
Visitor	Caller
Concrete Visitor	Campaigner
Element	Resident
Concrete Elements	HighIncome, LowIncome, MediumIncome
ObjectStructure	*Not implemented*
visitConcreteElement()	visit()
accept()	accept()
Client	main()
operation()	incomegroup()

Visitor

- The `Caller` class act as the visitor.
- The definition of the `visit()` method is overloaded to provide the functionality to visit the designated concrete elements (`HighIncome, MediumIncome` and `LowIncome`)

Concrete Visitor

- The `Campaigner` class acts as a concrete visitor.
- All variations of the `visit()` method can be implemented. It is possible to implement empty methods or rely on default implementations of the`visit()` function in cases where a visitor should not visit certain elements.
- It is easy to alter/add/remove concrete visitors from the system with no need to recompile the client or any of the elements in the object structure.

Element

- The `Resident` class acts as the element.
- The definition of the `accept()` method is polymorphic and allows for different implementations in `HighIncome, MediumIncome` and `LowIncome`.
- The `incomegroup()` method represent methods that are defined in the `Element` participant of the Vistor pattern. This method is typically called in the body of the `visit()` method that is implemented in the concrete visitors.

Concrete Element

- The `LowIncome, MediumIncome` and `HighIncome` classes act as the concrete elements.
- The implementtion of their respective `accept()` methods is the second dispatch of the application of double dispatch. It calls the `visit()` method of the `Caller` it received as parameter. The implementation of this method that should appear in each concrete `Resident` is shown below.
- The `incomegroup()` method represent methods that are defined in the `Element` participant of the Vistor pattern. This method is typically called in the body of the `visit()` method that is implemented in the concrete visitors.

The following code is the implementation of the `accept()` method that should appear in each concrete `resident`.

```
void accept(Caller& v) {
    v.visit(this);
}
```

Although the code is identical in each of these classes they are differnt from a compiler poit of view because the parameter that is passed in its parameter is of a different type in each of these cases.

In a typical execution the client calls the `accept()`-method of a concrete `Resident` and passes a pointer to a specific `Caller` as parameter. The `accept()`-method will then call

the `visit()`-method of the concrete `Caller` that was passed as parameter and passes a pointer to itself to this `Caller`. Lastly the `visit()`-method will call back to the concrete `Resident` by executing (in this case) its `incomegroup()`-method.

References

[1] Judith Bishop. *C# 3.0 design patterns*. O'Reilly, Farnham, 2008.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1994.

[3] Vince Huston. Design patterns. `http://www.cs.huji.ac.il/labs/parallel/Docs/C++/DesignPatterns/`, n.d. [Online: Accessed 29 June 2011].

[4] Jens. Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*, pages 9 –15, aug 1998.

[5] Wikipedia. Visitor pattern — wikipedia, the free encyclopedia, 2012. URL `\url{http://en.wikipedia.org/w/index.php?title=Visitor_pattern&oldid=516592783}`. [Online; accessed 21-October-2012].