Visitor Design Pattern [Gamma et al]

This pattern consists of two parts i.e. visitor and visitable and allows the encapsulation of polymorphic behavior outside the class hierarchy on which it operates. This pattern deals with classes that focus only on the data structures without knowing the logic that will be applied to the structure. At the same time, classes that provide concentrate solely on the logics that will be applied to the structure without knowing what the structure looks like. The benefit is that the evolution of the logics and the structures can vary independently. Following are the variants of the visitor pattern:

Visitor Combinators [74]

This implementation consists of a visitor interface that can be used to compose new visitors from given ones. The set of Combinators are proposed which includes traversal Combinators that can be used to obtain full traversal control. Combinators are reusable classes capturing basic functionality that can be composed in different constellations to obtain new functionality. The Visitor interface declares a visit method for each syntax rule in the grammar. The Visitable interface declares the accept method. This method applies the appropriate visit method from its argument visitor to the current top node. Iteration over a tree can either be implemented in the accept methods, or in default implementations of the visit methods. Following is the UML Diagram:

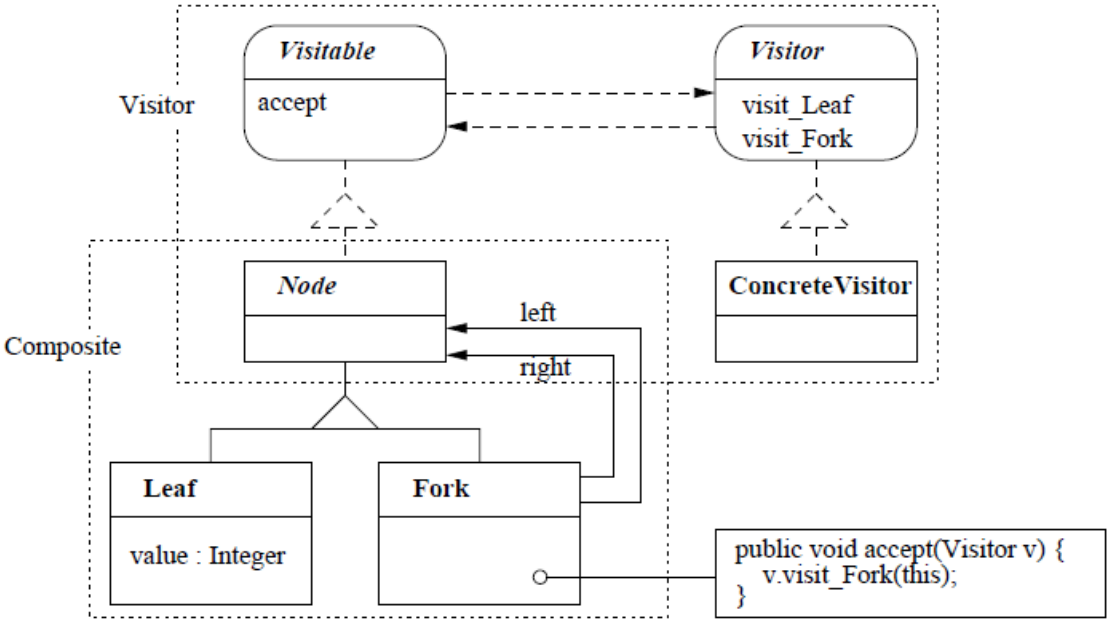

Figure 1.29 Visitor combinatory

Distributed Monitoring using Visitor Pattern [75]

This variant addresses the problem by taking out the cross-cutting operation from the components as a class called visitor and having the class visit the components. The components accept the visitor and let

it carry out the desired operation. In this way, adding a new operation becomes easy by simply adding a new visitor class. This makes models greatly flexible: Following is the UML:

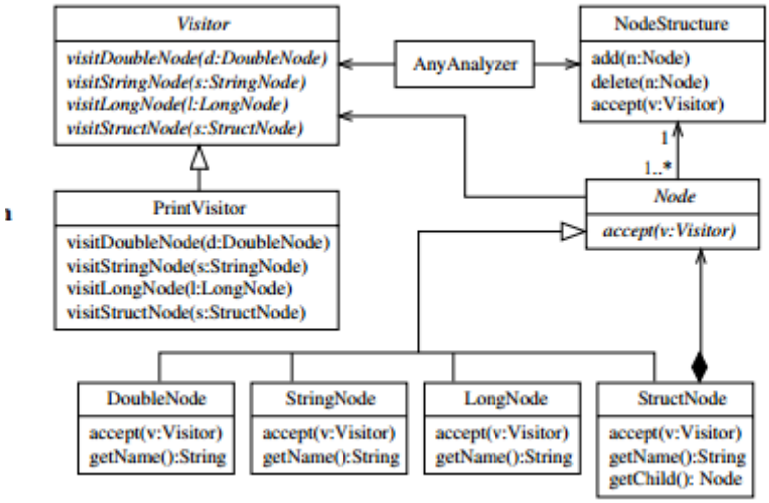

Figure 1.30 Visitor Variant

Extended Visitor Pattern [76]

In this variant, the technique was to create object-oriented predictive recursive descent parser by starting with an LL(1) context-free grammar and applying a simple transformation. The resulting equivalent grammar was directly modeled by a class structure using inheritance to represent branches and composition to represent sequences. Since the tokens determine whether or not the input corresponds to the grammar, a variant of the visitor design pattern was used to provide direct dispatching to the appropriate parsing code, thus eliminating conditions, and to allow the open-ended addition of tokens to the grammar with minimal perturbation of the existing code. Following is the UML Diagram of extended visitor pattern:

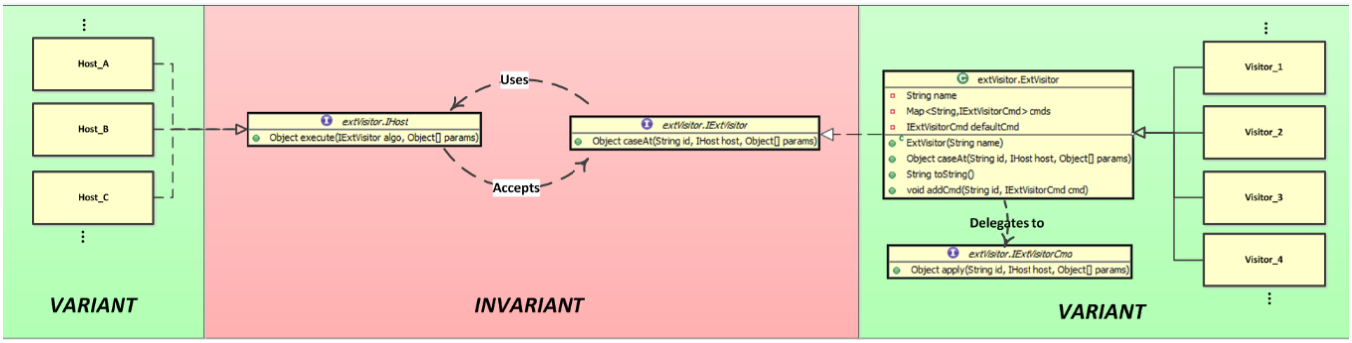

Figure 1.31 Extended Visitor Pattern