SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–35
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

A Survey Report of Enhancements to the Visitor Software Design Pattern

Tanumoy Pati*and James H. Hill

Dept. of Computer and Information Science, Indiana University-Purdue University Indianapolis, Indianapolis, IN USA
Email: {tpati, hillj}@cs.iupui.edu

SUMMARY

The Visitor pattern is a behavioral software design pattern where different objects represent an operation to be performed on elements of an object structure. Despite the benefits of the Visitor pattern, its rigid structure has limitations. Owing to the Visitor pattern's usefulness and importance to software design, many researchers have extended and modified the original Visitor pattern to overcome its limitations. Researchers have even replaced the Visitor pattern with more refined design patterns (*e.g.*, Reflective Visitor Pattern, Java Walkabout Class, and Dynamic Dispatcher) that bear minimal resemblance to the original Visitor pattern's structure while retaining its major advantages (*e.g.*, ability to add new operations to an object structure without changing the classes of objects, localizing related behavior, and accumulating state).
This article therefore provides a comprehensive survey of the Visitor pattern for software practitioners. Within the survey, we focus on major enhancements that have been made to the original Visitor pattern to overcome its limitations. Based on our survey results, we found that variations of the Visitor pattern can be separated into two categories: *extended* Visitor patterns where the original Visitor pattern structure stays intact, and *alternative* Visitor patterns where the structure of the original Visitor pattern is altered.

Received . . .

KEY WORDS: design patterns, visitor pattern, survey, extended visitor patterns, alternative visitor patterns

1. INTRODUCTION

The software industry demands effective solutions that support rapid development of software systems [1]. Object-oriented software systems inherently require frequent modification, extension, and correction [2]. This makes it essential to develop software systems using reusable abstractions that are clean (*i.e*, efficient, readable, well-structured, and simple) and easy to maintain.

Software design patterns, as put forth by the Gang of Four (GoF) [3], are effective, adaptable, reusable, and time-tested solutions to commonly recurring design problems. Each software design pattern systematically names, explains, and evaluates an important and recurring design challenge in object-oriented software systems [3]. Knowledge of software design patterns enhances a software practitioner's ability to construct clean and easily maintainable software designs [4].

The Visitor pattern [3] is one of the GoF's software design patterns that separates an algorithm from the object structure it operates on. It is also a behavioral pattern that provides a way to define new operations on an object structure without the need to change the object's classes. This pattern has been notably used in tools such as Java Tree Builder (JTB) [5], JJtree [6], and the Generic Modeling Environment (GME) [7].

*Correspondence to: tpati@cs.iupui.edu

Since Gamma et al. originally published the Visitor pattern circa 1995, numerous researchers (*e.g.*, Yun Mai et al. [8], Joost Visser [9], and Jens Palsberg et al. [10]) have proposed several extensions and variations to the original Visitor pattern. The purpose of the proposed extensions and variations was to improve the original Visitor pattern's functional limitations. For example, Yun Mai et al. [11] classified and organized the different variations of the Visitor pattern to assist software practitioners with selecting one that best suits their needs. Since Mai's classification, the Visitor pattern has been extended, modified, and even replaced with other design approaches. At present—to the best of the author's knowledge—there exists no article that documents the most recent extensions and variations to the original Visitor pattern.

This article therefore provides a comprehensive survey of the Visitor pattern for software practitioners. The main contributions of this article are as follows:

- It highlights the major enhancements that have been made to the original Visitor pattern since its introduction;
- It categorizes each enhanced Visitor pattern into either of two groups (*i.e.*, extended or alternatives) based on its degree of variation in maintaining structural integrity of the original Visitor pattern;
- It discusses the benefits of each enhanced or alternative Visitor pattern has over the original Visitor pattern;
- It discusses limitations of the extended or alternative Visitor patterns, if applicable.

Based on our survey results, we have learned that the extended and alternative Visitor patterns are applicable to certain scenarios, which we discuss in Section 5. The applicable scenarios, however, imply that it is hard to remove all of the original Visitor patterns' limitations without compromising some aspect of the software's design and performance.

Article organization. The remainder of this article is organized as follows: Section 2 provides an overview of the original Visitor pattern and lists its benefits and limitations; Section 3 discusses different extensions to the original Visitor pattern; Section 4 discusses different alternatives to the original Visitor pattern; Section 5 presents the scenarios for when a software practitioner should select an extension or alternative; and Section 6 provides the concluding remarks.

2. OVERVIEW OF ORIGINAL VISITOR PATTERN

The Visitor pattern represents an alternative approach to using object-oriented concepts on hierarchical structures where operations are separated from object structure. This simplifies extending the former (*i.e.*, the operations) without affecting the latter (*i.e.*, the object structure). For example, if the user wants to define a new operation and the functionality of the software is defined over many classes, then each class must be changed in order to accommodate the new operation. If the user implements the Visitor pattern, then the new operation can be added over an object structure by merely adding a new visitor subclass that represents the new operation.

Figure 1 illustrates the original Visitor pattern. As shown in this figure, the major participants (along with their description) in the Visitor pattern are as follows:

1. **Visitor.** The Visitor participant is a base class that declares a visit operation for each concrete element that it can visit. In Figure 1, the `Visitor` base class defines the `VisitConcreteElementA` and `VisitConcreteElementB` visit operations. This means that the Visitor is able to visit ConcreteElementA and ConcreteElementB objects, respectively. By default, each visit method's implementation is empty. This means that the Visitor subclasses need to only overload visit methods of interest.
2. **ConcreteVisitor.** The ConcreteVisitor participant is a subclass of the Visitor participant or another ConcreteVisitor participant. The ConcreteVisitor participant is responsible for implementing the visit method for each ConcreteElement participant that it can visit. As shown in Figure 1, `ConcreteVisitor1` and `ConcreteVisitor2` are ConcreteVisitor participants derived from the `Visitor` base class.

DOI: 10.1002/spe

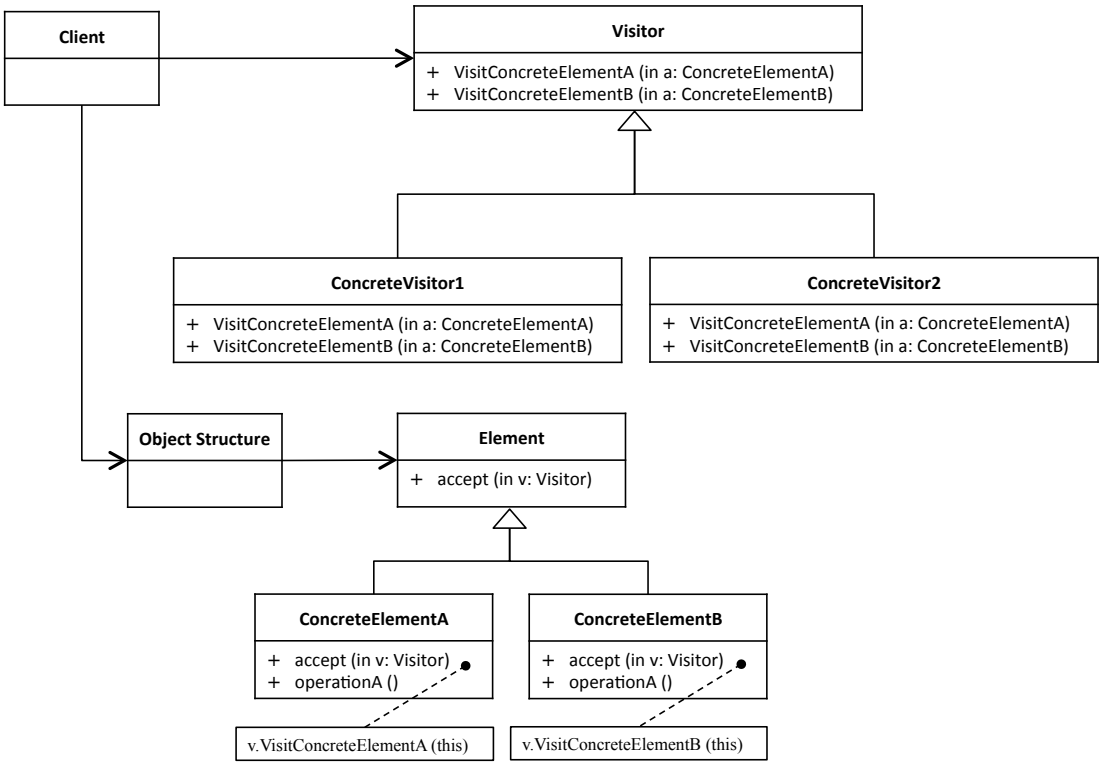

Figure 1. Structure of original Visitor software design pattern.

3. **Element.** The Element participant is an abstract class that defines an `accept()` method. The `accept()` method takes a Visitor object as a single parameter, which denotes the target visitor of the operation.
4. **ConcreteElement.** The ConcreteElement participant is a subclass of the Element participant. Each ConcreteElement participant is responsible for implementing the `accept()` method, which invokes the appropriate visit method on the Visitor participant. As shown in Figure 1, both `ConcreteElementA` and `ConcreteElementB` inherit from the `Element` participant and invoke `VisitConcreteElementA` and `VisitConcreteElementB`, respectively.

Lastly, the visit methods of the ConcreteVisitor participant are linked to their respective ConcreteElement participants. This coupling causes double-dispatch behavior in conventional single-dispatch object-oriented languages such as C++, Java, and Smalltalk [12].

```
1  /**
2   * Base class of all visitable objects.
3   */
4  class Object {
5  public:
6    virtual void accept (Visitor * v) = 0;
7  };
8
9  class Patron : public Object {
10 public:
11   int getPatronId (void) const { return this->patron_id_; }
12   virtual void accept (Visitor * v) { v->visitPatron (this); }
13
14   // other methods
15 private:
16   int patron_id_;
17 };
```


Prepared using speauth.cls
DOI: 10.1002/spe

```
18
19  class Book : public Object {
20  public:
21    string getBookName (void) const { return this->book_name_;  }
22    virtual void accept (Visitor * v) { v->visitBook (this); }
23
24    // other methods
25  private:
26    string book_name_;
27  };
28
29  /**
30   * Base class implementation of all visitor objects.
31   */
32  class Visitor {
33  public:
34    virtual void visitPatron (Patron *) { }
35    virtual void visitBook (Book *) { }
36  };
37
38  class PrintVisitor : public Visitor {
39  public:
40    void visitPatron (Patron * e) { cout << "Patron id:" << e->getPatronId (); }
41    void visitBook (Book * e) { cout << "Book name:" << e->getBookName (); }
42  };
43
44  //
45  // Main entry point of the application
46  //
47  int main (int argc, char * argv []) {
48    // Manually create an Patron
49    unique_ptr <Object> obj (new Patron ());
50
51    // Print the object's information.
52    PrintVisitor v;
53    obj->accept (&v);
54
55    return 0;
56  }
```

Listing 1: Source code snippet illustrating usage of the original Visitor pattern.

Listing 1 provides example C++ source code that illustrates the Visitor pattern's usage. As shown in this listing, the `Object` class denotes the Element participant, which is the base class for ConcreteElement participants `Patron` and `Book`. The Visitor participant is represented by the `Visitor` class, which is the base class for the ConcreteVisitor participant named `PrintVisitor`. The `PrintVisitor` class contains `visitPatron()` and `visitBook()` methods, which are responsible for visiting the ConcreteElement participants `Patron` and `Book`, respectively.

The advantages of the Visitor pattern—as originally listed by the Gang of Four—are as follows [3]:

1. The Visitor pattern enables users to add new operations to an object structure without changing the classes of the objects. For example, in Listing 1 new operations (*e.g.*, listing books borrowed by a patron) can be added by simply defining a new concrete visitor (*e.g.*, `BorrowedBookVisitor`).
2. The Visitor pattern localizes related behavior in the Visitor object, thereby separating related operations from unrelated ones. For example, in Listing 1 all behavior in `PrintVisitor` is related to printing information about each ConcreteElement participant. Any other behavior not related to printing information about each ConcreteElement participant would not appear in the `PrintVisitor` class.
3. The Visitor pattern helps with iterating over elements that do not share a common parent in the hierarchy. For example, in Listing 1 if there is a ConcreteElement participant named

DOI: 10.1002/spe

`Magazine` that is not in the `Object` hierarchy, then the `PrintVisitor` can still visit the ConcreteElement participant named `Magazine` even though the `Magazine` class does not share a common parent with the other ConcreteElement participants. The `Magazine` class only needs to implement an `accept()` method, and the Visitor participant needs a corresponding visit method for the `Magazine` ConcreteElement participant.

4. The Visitor pattern allows Visitor objects to accumulate state as it visits each element in the object structure. For example, in Listing 1 if there exists a `CountVisitor` that counts the number of books borrowed by all patrons, then the `CountVisitor` must store the number of books each patron borrowed as it iterates over all the patrons in the model.
5. The Visitor pattern helps with conforming to the open/closed principle [13], which states that software entities (*e.g.*, classes, functions, and modules) should be open for extension, but closed for modification. This is because the Visitor pattern allows addition of new operations without changing the classes of the elements on which it operates.

Likewise, the limitations of the Visitor pattern are as follows:

1. **Prior knowledge of the arguments and return type.** For example, in Listing 1 consider the user requires data to be returned from the object structure visitation. This new requirement makes it necessary to define new `visit()` methods that either return a new datatype (*e.g.*, string or integer) or receive extra arguments. A new visitor class therefore has to be defined along with providing new `accept()` methods to satisfy this requirement. This is a limitation because as new requirements are introduced into the code that require evaluation functions to receive data from a user or return data, the developer has to make sure that there exists a corresponding visitor class and `accept()` method. See Sections 3.6, 4.1, 4.2, and 4.5 for modifications to the original Visitor pattern that address this limitation.
2. **Adding new elements is difficult.** This is because each new Element participant added to the hierarchy requires a corresponding visit method to be added to the Visitor participant—and all the ConcreteVisitor participants. For example, consider an element structure consisting of numbers as nodes (*i.e.*, each number is a ConcreteElement participant) and the operations on those numbers represented as ConcreteVisitor participants. In such a scenario, each time a node is added to the number hierarchy, it is necessary for all the ConcreteVisitor participants and the `Visitor` base class to be updated for accommodating the new Element participant. See Sections 3.4, 4.1, 4.2, and 4.3 for modifications to the original Visitor pattern that address this limitation.
3. **Necessity of accept method.** The Visitor pattern requires each visitable ConcreteElement participant to have a *so-called* `accept()` method. For example, in Listing 1 the `Visitor` object is dependent upon the `accept()` method dispatching to the correct ConcreteElement, which in turn invokes the correct visit method. This is a limitation because the `accept()` method adds an extra level of indirection (*i.e.*, double-dispatch) that introduces a performance penalty and increases coupling [10]. See Sections 4.1, 4.2, 4.3, and 4.4 for modifications to the original Visitor pattern that address this limitation.
4. **Partial visitation.** The Element participants depend on the Visitor participant via the parameter in the `accept()` method. Likewise, the Visitor participant depends on the ConcreteElement participant. This cyclic dependency created by the Visitor pattern makes it necessary to override all Element-specific visit methods in every ConcreteVisitor. This, however, is a design limitation for ConcreteVisitor participants that want to implement only a subset of all the visit methods. See Sections 3.2 and 4.4 for modifications to the original Visitor pattern that address this limitation.
5. **Structure extension requires regeneration of traversal code.** The Visitor pattern requires regeneration of traversal code upon structure extension. If the traversal algorithm is embedded in the Visitor participant, then each ConcreteVisitor participant will inherit the same traversal code [3]. Whenever the object structure is extended, the traversal code in all the visitors must be updated accordingly. This is a limitation because similar traversal code in visitors adds a significant amount of redundancy to the source code. See Section 3.3 for modifications to the original Visitor pattern that address this limitation.

Prepared using speauth.cls
DOI: 10.1002/spe

6. **Little traversal control.** The Visitor pattern offers little traversal control when used for tree traversals [9,14]. This is a limitation because the original Visitor pattern enables encapsulation of polymorphic behavior, such as tree traversal outside the class hierarchy that it operates. This requires hard-wiring the traversal strategy into the `accept()` methods to allow little traversal control. Moreover, the original Visitor pattern also lacks the capability of hierarchical navigation and conditional navigation (*i.e.*, skipping branches during tree traversal). The lack of navigational capabilities can make the original Visitor pattern undesirable for designs that use the Composite pattern. See Sections 3.3, 3.5, and 4.5 for modifications to the original Visitor pattern that address this limitation.
7. **Implementation inheritance not supported.** In designs that do not use the Visitor pattern, *i.e.*, designs where operations are defined in the classes that they operate on, subclasses inherit all the methods of the base class—including those which they do not override. When applying the Visitor pattern, implementation inheritance has to be done manually because it is necessary for all the ConcreteVisitors to override all the ConcreteElement specific visit methods that exist in the Visitor interface. This is a limitation because implementation inheritance—when done manually—is difficult, time-consuming, and error-prone [15].
   For example, consider the class diagram shown in Figure 2a. As shown in this figure, `Herbivore` is derived from `Animal`, and `Pet` is derived from `Herbivore`. The `Animal` class implements two methods: `Diet()`, which `Herbivore` and `Pet` overload, and `Habitat()`, which only `Herbivore` overloads. Figure 2b illustrates applying the Visitor pattern to the class hierarchy shown in Figure 2a. As shown in Figure 2b, `HabitatVisitor` has to implement the `visitPet()` method even though the `Pet` class does not override the `Habitat()` method. The `HabitatVisitor` class has to either redirect the call to its `visitHerbivore()` method, or implement the same logic that appears in its implementation of `visitHerbivore()` method. See Sections 3.3 and 4.3

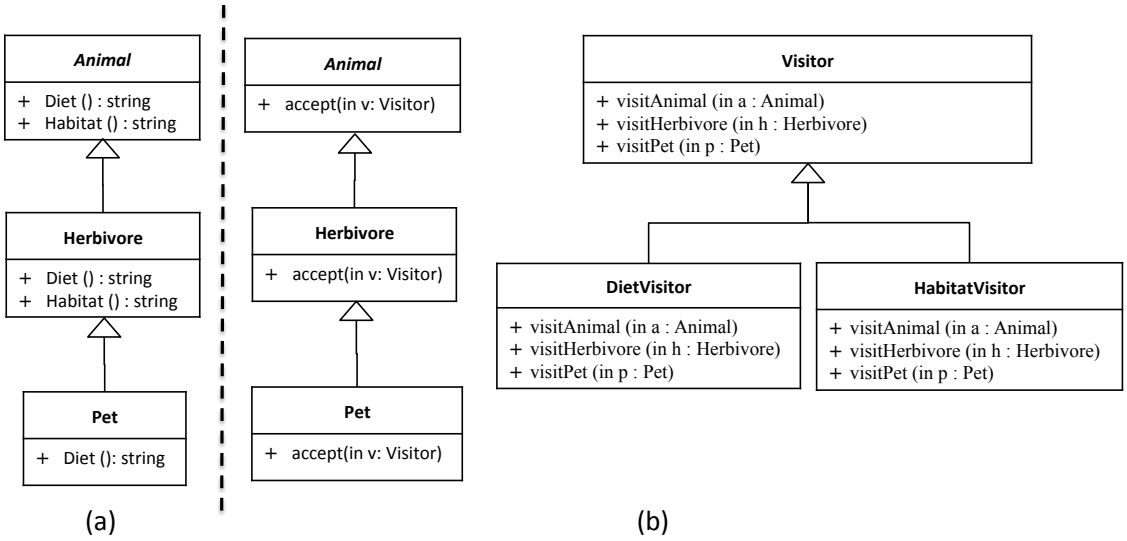

Figure 2. (a) Element hierarchy for Animal and (b) application of the original Visitor pattern to Animal element hierarchy to illustrate implementation inheritance is not possible using the original Visitor pattern.

for modifications to the original Visitor pattern that address this limitation.

8. **Violation of DIP and ISP.** The Visitor pattern violates the *dependency inversion principle* [16], which states that high-level modules and low-level modules should be independent of each other and should depend on abstractions. For example, in Listing 1 the parameters of the visit methods in the abstract Visitor class (higher-level module) are dependent on ConcreteElement participants (lower-level modules) (*i.e.*, `Book` and `Patron`). Moreover, the Visitor participant has more than one interface, which makes it necessary to

Prepared using speauth.cls
DOI: 10.1002/spe

modify all ConcreteVisitor participants each time the object structure changes. The Visitor pattern also violates the *interface segregation principle* [17], which states that once an interface becomes too big, it should be segregated into several smaller, specific interfaces so that users are not forced to depend on unused methods [18]. This is a limitation because violating DIP and ISP causes the Visitor pattern to violate basic principles of object-oriented programming and design [19]. See Section 3.4 for modifications to the original Visitor pattern that address this limitation.

9. **Inability to access the special functions of the ConcreteVisitor.** Although clients invoke the `accept()` method with a ConcreteVisitor participant, the `accept()` method's parameter is a Visitor base class. This is a limitation because it prohibits the `accept()` method from accessing any methods defined by the ConcreteVisitor participant. See Section 3.1 and 4.3 for modifications to the original Visitor pattern that address this limitation.
10. **Breaking encapsulation.** The Visitor pattern often forces the ConcreteElement to expose its internal state (*i.e.,* data) through public methods. This is because the visitors often require data kept by the visitable objects. This, however, is a limitation because it breaks encapsulation—a fundamental principle of object-oriented design and programming. See Section 4.5 for modifications to the original Visitor pattern that address this limitation.

Although the Visitor pattern has many limitations, it is important to software design. Software developers therefore have discovered many different techniques for addressing the limitations discussed above instead of disregarding the Visitor pattern. The remainder of this article discusses the results of a comprehensive literature review on how the original Visitor pattern's limitations have been addressed.

3. EXTENSIONS TO THE ORIGINAL VISITOR PATTERN

This section discusses different extensions to the original Visitor pattern to overcome some of the limitations introduced in Section 2. As discussed in Section 2, the structure of the original Visitor pattern consists of three critical parts:

- The visit methods on the Visitor and ConcreteVisitor participant that operate on the corresponding ConcreteElement participant;
- The `accept()` method on the Element and ConcreteElement participant that invokes the corresponding visit method on the Visitor participant; and
- The relationship between the Visitor and ConcreteVisitor participant where the Visitor participant defines different visit methods that can be selectively implemented by the ConcreteVisitor participant and the relationship between the Element and ConcreteElement participant where the Element defines an abstract `accept()` method that each ConcreteElement must implement.

The extended Visitor patterns therefore are *extensions* of the original Visitor pattern such that the structure of the original Visitor pattern—as described above—remains intact.

Table I gives a brief overview of different extensions to the original Visitor pattern that are discussed throughout the remainder of this section. As shown in this table, different extensions address different limitations of the original Visitor pattern. The remainder of this section discusses each of the extensions presented in Table I in more detail using the following format: the problem the extension addresses, its solution approach, its implementation, and known benefits and limitations.

3.1. Extended Types Visitor Pattern

Problem. In the original Visitor pattern, the client invokes the `accept()` method with a ConcreteVisitor element. The `accept()` method, however, receives the ConcreteVisitor participant as a Visitor participant. For example, in Listing 1 the client invokes the Patron's `accept()` method with a `PrintVisitor` object, which is a ConcreteVisitor participant. The

Prepared using speauth.cls
DOI: 10.1002/spe

Table I. Summary of extensions to the original Visitor pattern and the limitation each extension addresses.

Pattern Name	Limitations Addressed	Section Discussed
Extended Types Visitor Pattern	Inability to access special methods of the ConcreteVisitor	3.1
Acyclic Visitor Pattern	Partial visitation	3.2
Visitor Combinators Visitor Pattern	Structure extension requires regeneration of traversal code	3.3
	Little traversal control	
	Implementation inheritance not supported	
Normal Form Visitor Pattern	Adding new elements is difficult	3.4
	Violation of DIP and ISP	
Hierarchical Visitor Pattern	Little traversal control	3.5
Generic Visitor Pattern	Prior knowledge of the arguments and return type	3.6

Patron's `accept()` method, however, views the `PrintVisitor` object as a `Visitor` object. This prevents the `accept()` method from accessing methods implemented by `PrintVisitor` that are not defined in the `Visitor` base class.

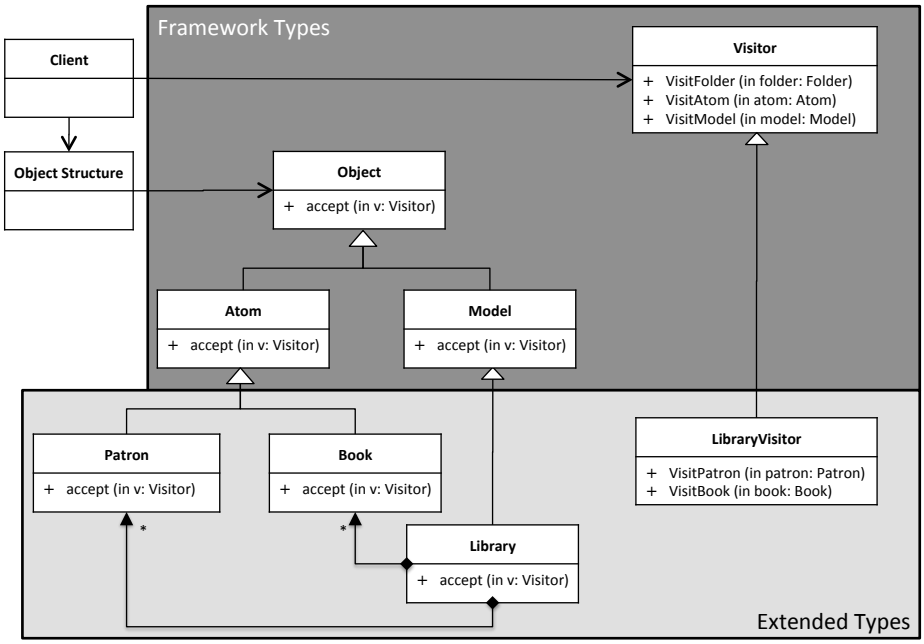

Figure 3. A sample design highlighting the use of Visitor pattern in the modeling world.

Another scenario where the Visitor pattern fails is illustrated in Figure 3. This figure illustrates a example design consisting of two element hierarchies: (1) framework types and (2) extend types from a Library domain. The `Visitor` base class does not recognize the extended types (*i.e.*, Book, Library, and Patron) and—owing to the open/closed principle—modification of the `Visitor` base class is not allowed. The `LibraryVisitor` therefore inherits from the `Visitor` base class so the client can visit the extended types. The `Library`, `Book` and `Patron` class, however, must implement the `accept()` method in terms of the `Visitor` base class, and not the `LibraryVisitor` extended class. Unfortunately, the `Visitor` object does not recognize the extended types and the `visit()` methods for the base types are invoked instead of the `visit()` methods for the extended types.

DOI: 10.1002/spe

Solution Approach. The use of *dynamic type casting* in the `accept()` method of individual classes is an effective solution to the above mentioned drawback [10, 20]. Dynamic binding allows the client to access other functions in the derived class. For the scenario presented in Figure 3, dynamic type casting ensures that the extended visitor (*i.e.*, `LibraryVisitor`) invokes the appropriate visit method for its corresponding extended type. In case of faulty extended types or a bad typecast, the visit method for the corresponding base type is invoked (*i.e.*, a fall-back solution). A similar approach has also been used in the visit method in an attempt to use Visitor pattern in frameworks [21].

Implementation. Sample code of the Extended Types Visitor Pattern is shown in Listing 2. As shown in this listing, the `accept()` method in the `Patron` (line 6) and `Book` (line 24) class take a `Visitor` parameter. The `accept()` method then attempts to dynamic cast (line 8 and 26) the generic visitor to a more concrete visitor (*i.e.*, `LibraryVisitor`). If the dynamic cast succeeds, then the `accept()` method invokes the appropriate visit method only known to the `LibraryVisitor` object. Otherwise, the `accept()` invokes the appropriate visit method on the `Visitor` object (line 13 and 31).

```
1  class Patron : public Atom {
2  public:
3    int getPatronId (void) const { return this->patron_id_; }
4    // other methods
5
6    virtual void accept (Visitor * v) {
7      // Dynamic cast to extended types visitor
8      LibraryVisitor * lv = dynamic_cast <LibraryVisitor *> (v);
9
10     if (0 != lv)
11       lv->visitPatron (this);
12     else
13       v->visit (this);
14   }
15
16 private:
17   int patron_id_;
18 };
19
20 class Book : public Atom {
21 public:
22   const string & getBookName (void) const { return this->book_name_; }
23
24   virtual void accept (Visitor *v) {
25     // Dynamic cast to extended types visitor
26     LibraryVisitor * lv = dynamic_cast <LibraryVisitor *> (v);
27
28     if (0 != lv)
29       lv->visitBook (this);
30     else
31       v->visit (this);
32   }
33
34 private:
35   string book_name_;
36 };
37
38 class LibraryVisitor : public Visitor {
39 public:
40   virtual ~LibraryVisitor (void) { }
41   virtual void visitPatron (Patron * e) { }
42   virtual void visitBook (Book * e) { }
43 };
44
45 class PrintVisitor : public LibraryVisitor {
46 public:
47   virtual ~PrintVisitor (void) { }
48   void visitPatron (Patron * e) { cout << "Patron id:" << e->getPatronId (); }
```


DOI: 10.1002/spe

```
49     void visitBook (Book * e) { cout << "Book name:" << e->getBookName (); }
50  };
```

Listing 2: Source code snippet illustrating usage of the Extended Types Visitor pattern.

Benefits. The following is a list of benefits for the Extended Types Visitor Pattern:

- It provides more flexibility to the client by allowing access to all the methods defined on the ConcreteVisitor.
- It guarantees that the extended Visitor calls the corresponding visit method for an extended type instead of the corresponding visit method for the base type of the extended type. In a more general sense, the extended Visitor calls the appropriate visit method for the ConcreteElement participant and not the Element participant.

Limitations. The following is a list of limitations for the Extended Types Visitor Pattern:

- The dynamic cast in the `accept()` method results in a runtime type check. This adds overhead to the application's performance [22].
- The Extended Types Visitor pattern is most suitable to use when an extended visitor is responsible for a unique set of extended elements. This is because the `accept()` method of the extended elements know about the extended visitor type—similar to how the base types know about the base Visitor type. Otherwise, every time a new visitor is added, the `accept()` method of all the ConcreteElement participants in the hierarchy has to be modified. This, in turn, leads to a chain of dynamic cast attempts in the `accept()` method, which adds more overhead to the application's performance. Moreover, it will add more coupling to the application's design.

3.2. Acyclic Visitor Pattern

Problem. The cyclic dependency between elements in the ConcreteElement hierarchy and the Visitor base class makes it necessary to modify the entire Visitor hierarchy every time a new ConcreteElement participant is added to the Element hierarchy. Moreover, this cyclic dependency makes it necessary for each ConcreteVisitor participant to have visit method for each of the ConcreteElement participants.

Solution Approach. Martin [23] provided a solution to this problem by designing the *Acyclic Visitor pattern* that uses multiple inheritance and dynamic type casting to break the cyclic dependency. The Acyclic Visitor pattern is similar to the Extended Types Visitor pattern (see Section 3.1), but its intent and implementation is slightly different as explained next.

Implementation. Figure 4 illustrates the participant diagram for the Acyclic Visitor pattern. As shown in this figure, the major participants in the Acyclic Visitor pattern are:

- **Element.** The Element participant is an abstract class that defines an `accept()` method. It is also the base class of the Element hierarchy.
- **ConcreteElement.** The ConcreteElement participants are the subclasses of the Element participant. `Element1` and `Element2` are examples of ConcreteElement participants in Figure 4.
- **Visitor.** The Visitor participant is a degenerate base class, *i.e.* it has no member functions. It acts as a placeholder in the type structure. The `accept()` method in the Element hierarchy takes this Visitor participant as its argument.
- **ElementVisitor.** The ElementVisitor participants are abstract visitors that correspond to each of the ConcreteElement participants. The ElementVisitor participants have a single abstract visit method that takes a reference to the ConcreteElement as its argument. `Element1Visitor` and `Element2Visitor` are examples of ElementVisitor participants in Figure 4 that are responsible for visiting `Element1` and `Element2`, respectively.
- **ConcreteVisitor.** The ConcreteVisitor is an actual visitor class that is derived from the Visitor and ElementVisitor participant that correspond to visitable ConcreteElement participants. `VisitForF` is an example of ConcreteVisitor participant in Figure 4.

DOI: 10.1002/spe

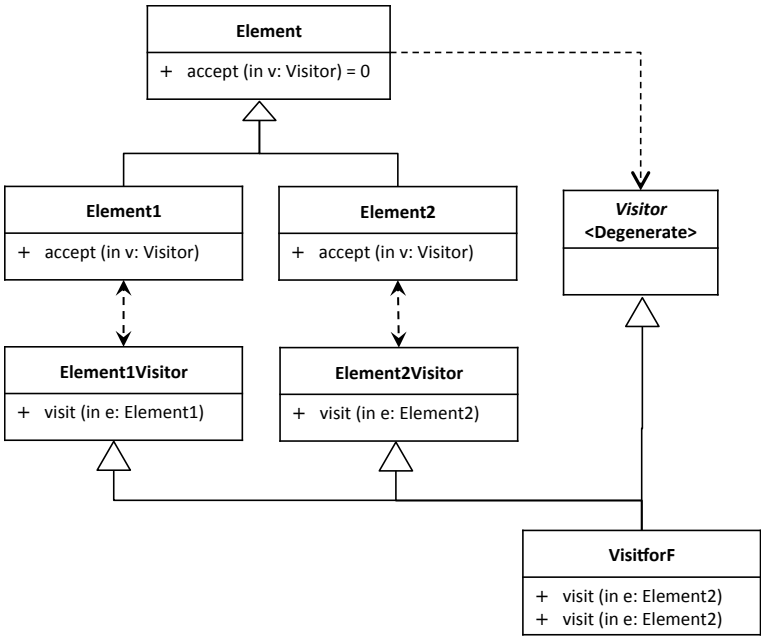

Figure 4. Class diagram of Acyclic Visitor pattern.

Listing 3 provides example implementation of the Acyclic Visitor Pattern. As shown in this listing, the `accept()` method of the ConcreteElement participants uses dynamic type casting (line 16 and 30) to cast the Visitor object to the appropriate ConcreteVisitor participant. If the dynamic type casting succeeds, then the corresponding ConcreteElement participant's visit method is invoked on the ConcreteVisitor. Otherwise, an exception is thrown back to the client.

```
1  class Visitor {
2  public:
3    virtual ~Visitor (void) = 0;
4  };
5
6  class Object {
7  public:
8    virtual void accept (Visitor * v) = 0;
9  };
10
11 class Patron : public Object {
12 public:
13   int getPatronId (void) const { return this->patron_id_; }
14
15   virtual void accept (Visitor *v) {
16     PatronVisitor * pv = dynamic_cast <PatronVisitor *> (v);
17
18     if (pv != 0)
19       pv->visitPatron (this);
20     else
21       throw invalid_cast ("Type-casting failed!");
22   }
23 };
24
25 class Book : public Object {
26 public:
27   const string & getBookName (void) const { return this->book_name_; }
28
29   virtual void accept (Visitor *v) {
30     BookVisitor *pv = dynamic_cast <BookVisitor *> (v);
31
32     if (pv != 0)
```


Prepared using speauth.cls
DOI: 10.1002/spe

```
33        pv->visitBook (this);
34      else
35        throw invalid_cast ("Type-casting failed!");
36    }
37
38  private:
39    string book_name_;
40  };
41
42  class PatronVisitor {
43  public:
44    virtual void visitPatron (Patron * e) = 0;
45  };
46
47  class BookVisitor {
48  public:
49    virtual void visitBook (Book * e) = 0;
50  };
51
52  class PrintVisitor : public Visitor, public PatronVisitor, public BookVisitor {
53  public:
54    void visitPatron (Patron * e) { cout << "Patron id:" << e->getPatronId(); }
55    void visitBook (Book * e) { cout << "Book name:" << e->getBookName(); }
56  };
```

Listing 3: Source code snippet illustrating usage of the Acyclic Visitor pattern.

Benefits. The following is a list of benefits for the Acyclic Visitor pattern, as highlighted by Martin [23]:

- The Acyclic Visitor pattern eliminates the cyclic dependency between Visitor and ConcreteElement participants. This minimizes recompilation of the Visitor and ConcreteVisitor participants when adding new ConcreteElement participants to the Element hierarchy.
- The Acyclic Visitor pattern allows partial visitation without the added expense of additional source code in the ConcreteVisitor participants.

Limitations. The following is a list of limitations for the Acyclic Visitor Pattern:

- The dynamic cast in the `accept()` method results in a runtime type check. This adds overhead to the application's performance and delays error detection from compile to run time [22].
- The developer must remember all the different types of ElementVisitor participants that are inherited by a ConcreteVisitor participant. Each time a new ConcreteElement is added to the Element hierarchy, the developers must remember what ConcreteVisitor participants must be updated. This process can be cumbersome for large element hierarchies.

3.3. Visitor Combinator Pattern

Problem. The original Visitor pattern enables encapsulation of polymorphic behavior, *e.g.*, tree traversal outside the class hierarchy on which it operates [9]. Because the traversal strategy is explicitly defined by either the `accept()` method or the ConcreteVisitor participant, it provides minimal traversal control to the client. Moreover, it is hard to combine visitors because—even with languages that support multiple inheritance—only one visit method is inherited from a parent class. This situation therefore restricts combination of visitors.

Solution Approach. Visser [9] provided a clean solution to the problem discussed above. He created a set of small, reusable classes called *visitor combinators* that capture the basic functionality of tree traversal control. The different visitor combinators can then be combined together in different arrangements to construct complex visitors that provide new functionality and enhance tree traversal control. Uniquely, the constructed visitors traversal strategy and applicability conditions are controlled by the visitor combinators.

Implementation. Table II lists the set of basic visitor combinators. As shown in this table, *identity combinator* is the default combinator that satisfies the `Visitor` class. Listing 4 provides a code

DOI: 10.1002/spe

Table II. The set of basic visitor combinators.

Combinator	Description
identity	Do nothing (non-iterating default visitor)
sequence(v1, v2)	Sequentially perform visitor v2 after v1
fail	Raise exception
choice(v1, v2)	Try visitor v1. If v1 fails, try v2 (left-biased choice)
all(v)	Apply visitor v sequentially to every immediate subtree
one(v)	Apply visitor v sequentially to the immediate subtrees until it succeeds

snippet that illustrates usage of the identity combinator. As shown in this listing, the `Identity` class is derived from the `Visitor` base class and overrides the visit methods of interest defined in the `Visitor` base class. The visit methods, however, are empty methods.

```
1  class Visitor {
2  public:
3    virtual void visitPatron (Patron *patron) { };
4    virtual void visitBook (Book *book) { };
5  };
6
7  class Identity : public Visitor {
8  public:
9    virtual void visitPatron (Patron *patron) { };
10   virtual void visitBook (Book *book) { };
11 };
```

Listing 4: Source code snippet illustrating usage of the identity combinator.

The *sequence combinator* is a binary combinator that applies two visitors sequentially to an Element participant. Listing 5 illustrates usage of the sequence combinator. As shown in this listing, the `Sequence` class is derived from the `Visitor` class and its constructor receives two Visitor objects as arguments. The visit methods (*i.e.*, `visitPatron` and `visitBook`) sequentially invoke the `accept()` method on the ConcreteElement participants (*i.e.*, `Patron` and `Book`) with the two visitors.

```
1  class Sequence : public Visitor {
2  public:
3    Sequence (Visitor *first, Visitor *second)
4      : first_ (first), second_ (second) { }
5
6    void visitPatron (Patron *patron) {
7      patron->accept (this->first_); patron->accept (this->second_);
8    }
9
10   void visitBook (Book *book) {
11     book->accept (this->first_); book->accept (this->second_);
12   }
13
14 private:
15   Visitor * first_, * second_;
16 };
```

Listing 5: Source code snippet illustrating usage of the sequence combinator.

The *choice combinator* applies the first visitor to the Element participant (*i.e.*, visits the Element using the first visitor). If the first visit fails, then the choice combinator applies the second visitor to the Element participant. Listing 6 illustrates usage of the choice combinator. As shown in this listing, the `Choice` class is derived from the `Visitor` class and its constructor receives two Visitor objects as arguments. The visit methods (*i.e.*, `visitPatron` and `visitBook`) first try to invoke the `accept()` method on ConcreteElement participants (*i.e.*, `Patron` and `Book`) using the Visitor named `first_` (line 8 and 17). If this visit fails, then the visit methods invoke the `accept()` method on ConcreteElement participants using the Visitor named `second_` (line 11 and 20).

DOI: 10.1002/spe

```
1  class Choice : public Visitor {
2  public:
3    Choice (Visitor *first, Visitor *second)
4      : first_ (first), second_ (second) { }
5
6    void visitPatron (Patron *patron) {
7      try {
8        patron->accept (this->first_);
9      }
10     catch (const VisitFailure &) {
11       patron->accept (this->second_);
12     }
13   }
14
15   void visitBook (Book *book) {
16     try {
17       book->accept (this->first_);
18     }
19     catch (const VisitFailure &) {
20       book->accept (this->second_);
21     }
22   }
23
24 private:
25   Visitor * first_, * second_;
26 };
```

Listing 6: Source code snippet illustrating usage of the choice combinator.

The *all combinator* is a traversal combinator that applies the specified visitor to all the subtrees of the starting node, *i.e.*, it traverses all the subtrees. Finally, the *one combinator* is a traversal combinator that applies the specified visitor to only one subtree. The one combinator first tries to apply the visitor to the left subtree. If the traversal fails, then it applies the visitor to the right subtree. The one combinator is different from the choice combinator because the one combinator involves a single visitor and the choice combinator involves more than one visitor. Due to the complexity of the all and one combinators, we do not provide any example source code. We, however, recommend referring to Visser's article [9] for examples of implementing the all and one combinators.

Benefits. The following is a list of benefits of using Visitor Combinator, as highlighted by Visser:

- The visitor combinators can create new ConcreteVisitor participants visitors from existing ConcreteVisitor participant to obtain new functionality.
- When visitors iterate over an object structure, they use their state to pass data instead of the call stack. As a result, at back-tracking, the visitor needs to explicitly restore the state, which is done automatically when using the call stack. The use of combinators eliminates the need to perform explicit stack maintenance while iterating the object structure. Instead, stack maintenance can now be done by separate, reusable combinators [9].
- The combinators are more robust when compared to the original Visitor pattern. Their reusability and full tree traversal control has enabled tools such as JJTraveler (`www.program-transformation.org/Tools/JJTraveler`) to implement generic visitor combinators for Java [24].

Limitations. The following is a list of limitations for the Visitor Combinator pattern:

- The set of combinators presented in Table II lack in generality. This is because every visitor must redefine all the visit methods—even if each visitor requires the same behavior [9]. Moreover, the combinator's implementation is specific to its context, which hinders its generality and reusability.
- Forwarding method calls between combinators adds extra levels of indirection, which increases performance overhead.

3.3.1. Generic Visitor Combinator Pattern. Visser [9] addressed the visitor combinator's generality problem by refactoring the design of the Visitor hierarchy such that syntax-specific functionality is

DOI: 10.1002/spe

separated from generic functionality. This solution is a variation on the staggered Visitor pattern [21] which introduced the following generic participants:

- **AnyVisitor.** The AnyVisitor participant is an abstract class that implements a `visitAny()` method. The purpose of the `visitAny()` method is to visit any ConcreteElement participant in the Element hierarchy.
- **AnyElement.** The AnyElement participant is an abstract class that implements an `acceptAny()` method. The purpose of the `acceptAny()` method is to accept any ConcreteVisitor participant as its argument. Moreover, for the generic `All` and `One` combinators, the AnyElement class declares two additional methods: *nrOfKids()*, which returns the number of children contained in a visitable node; and *getKid()*, which returns a single child element.

The AnyVisitor and AnyElement participants allow the basic visitor combinators (see Table II) to have syntax-independent definitions. For example, Listing 7 illustrates usage of the generic visitor combinators. As shown in this listing, the `Visitor` class implements the `visitAny()` method. The `Choice` class overrides this method and calls the `acceptAny()` method on the Element that is passed as an argument to the `visitAny()` method.

```
1  class AnyVisitor {
2  public:
3    virtual void visitAny (AnyElement *element) = 0;
4  }
5
6  class AnyElement {
7  public:
8    virtual void acceptAny (AnyVisitor * av) { av->visitAny (this); }
9    virtual int nrOfKids () = 0;
10   virtual Element * getKid (int i) = 0;
11 };
12
13 class Element : public AnyElement {
14 public:
15   virtual void accept (Visitor * v) = 0;
16 };
17
18 class Patron : public Element {
19 public:
20   virtual void accept (Visitor * v) { v->visitPatron (this); }
21   int nrOfKids (void) { return 2; }
22
23   Element * getKid (int i) {
24     switch (i) {
25       case 0: return leftchild;
26       case 1: return rightchild;
27       default: return 0;
28     }
29   }
30 };
31
32 class Book : public Element {
33 public:
34   virtual void accept (Visitor * v) { v->visitBook (this); }
35   int nrOfKids (void) { return 2; }
36
37   Element * getKid (int i) {
38     switch (i) {
39       case 0: return leftchild;
40       case 1: return rightchild;
41       default: return NULL;
42     }
43   }
44 };
45
46 class Visitor : public AnyVisitor {
```


DOI: 10.1002/spe

```
47  public:
48    void visitAny (AnyElement *element) {
49      Element *v = dynamic_cast <Element *> (element);
50
51      if (v != 0)
52        v->accept (this);
53      else
54        throw invalid_cast ("visit failed");
55    }
56
57    virtual void visitPatron (Patron * e) = 0;
58    virtual void visitBook (Book * e) = 0;
59  };
60
61  class Choice : public AnyVisitor {
62  public:
63    Choice (Visitor *first, Visitor *then)
64      : first_ (first), second_ (second) { }
65
66    void visitAny (AnyElement *x) {
67     try {
68       x->acceptAny (this->first_);
69     }
70     catch (const VisitFailure &) {
71       x->acceptAny (this->then_);
72     }
73
74  private:
75    Visitor * first_, * second_;
76  };
```

Listing 7: Source code snippet illustrating usage of the Generic Visitor Combinator pattern.

Visser also introduced a new combinator named the *forward combinator* that uses a generic visitor to implement all syntax-specific visit methods. Listing 8 illustrates usage of the forward combinator. As shown in this listing, the `Fwd` class implements the visit methods for each ConcreteElement, and forwards visits to the contained `AnyVisitor` (line 6 and 7).

```
1  class Fwd : public Visitor {
2  public:
3    Fwd (AnyVisitor * v)
4      : v_ (v) { }
5
6    void visitPatron (Patron *patron) { this->v_->visitAny (patron);  }
7    void visitBook (Book *book) { this->v_->visitAny (book); }
8
9  private:
10   AnyVisitor *v_;
11 };
```

Listing 8: Source code snippet illustrating usage of the forward combinator.

When generic combinators, *e.g.*, the choice combinator in Listing 7, are passed to the forward combinator, the original syntax-specific combinators are obtained. The consequences for this generality are an extra level of indirection and dynamic casting. The generic combinators, however, simplify the task of creating user-defined combinators. This is because the user-defined combinators depend only on those ConcreteElement participants that are relevant to the functionality the user-defined combinators implement.

3.4. Normal Form Visitor Pattern

Problem. In the original Visitor pattern, a change to a ConcreteElement leads to an increased number of objects that must be recompiled. Xiao-Peng et al. [18] highlighted that this occurs because the original Visitor pattern violates the *Dependency Inversion Principle (DIP)* [16]. The DIP states that higher-level modules should be independent of the lower-level modules. In reality,

DOI: 10.1002/spe

both higher-level and lower-level modules should depend on abstractions. In the original Visitor pattern, the visit methods depend on concrete classes (*i.e.*, the ConcreteElement participants).

The original Visitor pattern also violates the *Interface Segregation Principle (ISP)* [17]. The ISP states that an interface should be split into many client-specific interfaces when it becomes too big. The original `Visitor` has a large interface (*i.e.*, the Visitor base class) that is implemented by many ConcreteVisitor participants. When a single change occurs on the Visitor base class, then the change has to be propagated to all the ConcreteVisitor participants—even if a ConcreteVisitor participant is not interested in the visit method being modified. Moreover, the ConcreteElement participant depends upon the Visitor participant via the `accept()` method. Every time a new derivative of the ConcreteElement participant is created, the Visitor and ConcreteVisitor participant must change to incorporate the new ConcreteElement participant. This requirement therefore leads to an increased number of objects being recompiled.

Solution Approach. Xiao-Peng et al. used normal form theory [25] to improve the original Visitor pattern and conform it to the ISP and DIP. The use of normal form theory—a technique for determining a table's degree of vulnerability to logical inconsistencies and anomalies [25]—removed any transitive dependencies that existed in the original Visitor pattern. By removing transitive dependencies in the original Visitor pattern, it ensures the original Visitor pattern obeys the DIP.

Implementation. Listing 9 shows the functional dependencies of the original Visitor pattern that can be represented as *VisitorPattern (Visitor, Element, ConcreteElement, ConcreteVisitor)* [18]. As discussed by Xiao-Peng et al., the functional dependency $ConcreteElement \rightarrow Visitor$ violates the DIP because $ConcreteElement$ is a lower-level module and $Visitor$ is a higher-level module. As per the $VisitorPattern$ definition above, $Visitor$ is functionally dependent on $ConcreteElement$.

```
1  Visitor → Element;
2  Visitor → ConcreteVisitor;
3  Element → ConcreteElement;
4  ConcreteElement → Visitor;
```

Listing 9: Functional dependencies of the original Visitor pattern.

Figure 5 illustrates the participant diagram for the Normal Form Visitor pattern. As shown in this figure, `IVisitable` is an interface that declares an `accept()` method which takes a `VisitorFactory` object as an argument. ConcreteElement participants (*e.g.*, `VisitElement1` and `VisitElement2`) implement the IVisitable interface. `IVisitorFactory` is the interface that declares the `getVisitor()` method to return the visitor. The `VisitorFactory` participant implements this interface and additionally stores $< ConcreteElement, ConcreteVisitor >$ pairs using a hash map. The `VisitorFactory` participant then implements the `getVisitor()` method and returns the appropriate AbstractVisitor participant. Finally, the `AbstractVisitor` is the base class for ConcreteVisitor participants (*e.g.*, `ConcreteVisitor1` and `ConcreteVisitor2`).

To remove the functional dependency $ConcreteElement \rightarrow Visitor$, `IVisitorFactory` (see Figure 5) contains a method named `getVisitor()` that returns a Visitor participant for the corresponding Element participant. Likewise, the `accept()` method on the `IVisitable` interface has a `VisitorFactory` parameter. Owing to this design refactoring, the `Element` object is now dependent on `IVisitorFactory`, which breaks the functional dependency between $Visitor$ and $ConcreteElement$.

Listing 10 shows the functional dependencies of the Normal Form Visitor pattern that conform to the ISP and DIP. As shown in this listing, $Visitable$ (*i.e*, the lower-level module) is dependent on $IVisitorFactory$ (*i.e.*, the higher-level module). Likewise, $VisitorFactory$ (*i.e.*, the lower-level module) is dependent on $IVisitorFactory$ (*i.e.*, the higher-level module). $IVisitorFactory$ also conforms to ISP by splitting the design into multiple interfaces.

```
1  AbstractVisitor → VisitorFactory;
2  AbstractVisitor → ConcreteVisitor;
3  Element → ConcreteElement;
```


DOI: 10.1002/spe

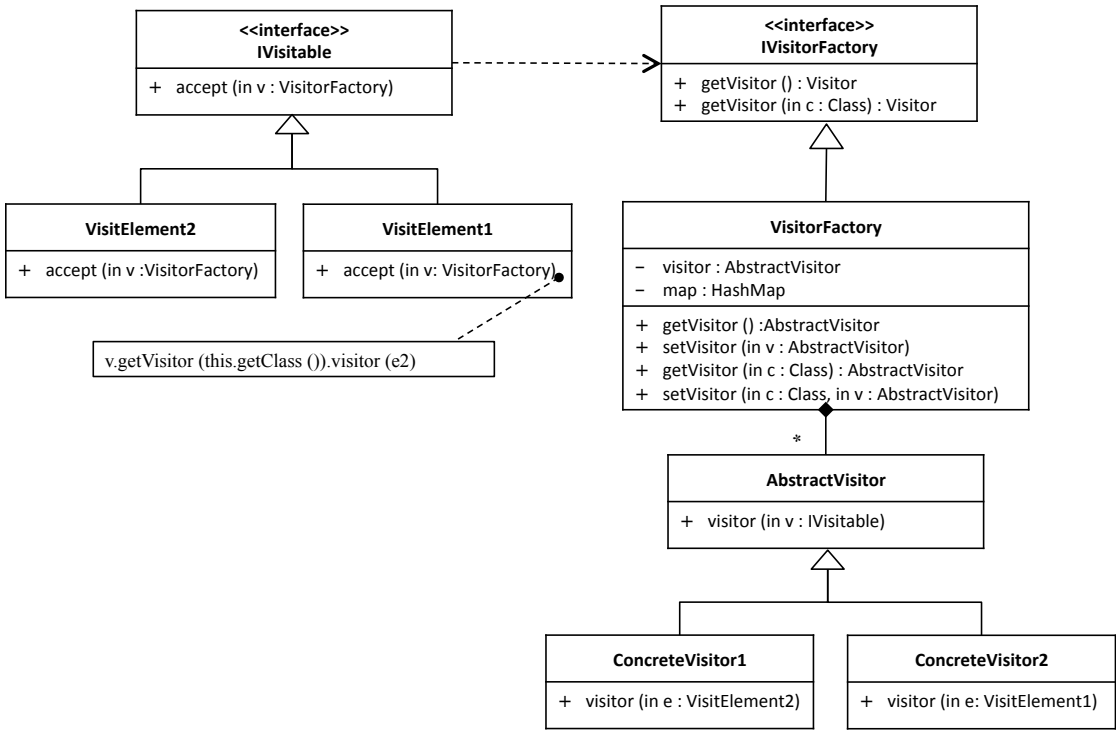

Figure 5. Class diagram of Normal Form Visitor pattern.

```
4  IVisitorFactory → Element;
5  IVisitorFactory → VisitorFactory;
```

Listing 10: Functional dependencies of the Normal Form Visitor pattern.

Benefits. The following is a list of benefits for the Normal Form Visitor pattern, as explained by Xiao-Peng et al.:

- The Normal Form Visitor pattern conforms to the DIP and ISP.
- The Normal Form Visitor pattern decreases the number of objects that must be recompiled when a ConcreteElement participant changes.
- The Normal Form Visitor pattern has high running efficiency and increased commonality as compared to the Acyclic Visitor pattern (see Section 3.2) and the Reflective Visitor pattern (see Section 4.1) [8].

Limitations. The following is a list of limitations for the Normal Form Visitor pattern:

- The use of a hash map to store $< ConcreteVisitor, ConcreteElement >$ pairings causes extra overhead because the ConcreteVisitor participant will contain visit methods for most of the ConcreteElement participants. This means that there will be multiple entries (or clashes) in the hash map for the same ConcreteVisitor participant against different ConcreteElement participants.

3.5. Hierarchical Visitor Pattern

Problem. The original Visitor pattern is incapable of providing navigational capabilities for designs that use the Composite pattern. For example, consider a file system where directories are composite nodes and files are leaf nodes. Likewise, the composite has two operations: *filepath*, which returns the full path of a file; and *filesearch*, which searches for a file using the full path of a file. In this example, the original Visitor pattern is not a good solution because of the following limitations:

Prepared using speauth.cls
DOI: 10.1002/spe

- **Hierarchical navigation.** The original Visitor pattern lacks the concept of depth in an object hierarchy. The Visitor pattern therefore cannot determine if one composite is derived from another composite, or if the composites are siblings.
- **Conditional navigation.** The original Visitor pattern does not allow skipping different branches during tree traversal. Traversals therefore cannot be optimized, or stopped based on conditions.

Solution Approach. Falco [26] presented a solution to the limitations discussed above named the *Hierarchical Visitor pattern*. The Hierarchical Visitor pattern has four major participants:

- **Component.** The Component is the base class common to Leaf and Composite participants. The Component also defines the `accept()` method.
- **Composite.** The Composite is the Component that contains children Component participants. The Composite also implements the `accept()` method that visits itself and its children.
- **Leaf.** The Leaf is a Component that contains no children. It also implements the `accept()` method that visits only itself.
- **HierarchicalVisitor.** The HierarchicalVisitor is the abstract Visitor that implements the `visitEnter()` method and `visitLeave()` method for the Composite participants, and the corresponding visit method for Leaf participants.

As the Hierarchical Visitor pattern implements the `visitEnter()` and `visitLeave()` methods, it overcomes the aforementioned navigation limitation. The Hierarchical Visitor pattern overcomes the conditional navigation limitation by making the return value of the `accept()` method boolean. If the `accept()` method returns false, then traversal stops at the current depth.

Implementation. Listing 11 provides an example implementation of the Hierarchical Visitor pattern. As shown in the listing, the `Directory` class represents a Composite component and the `File` class represents a Leaf component. The Visitor implemented in the listing constructs qualified path names for each `File` using the hierarchical navigational features of the Hierarchical Visitor Pattern. The abstract `Visitor` class declares virtual methods: `visitEnter()`, `visitLeave()` and `visit()`. The `visitEnter()` and `visitLeave()` methods take a `Directory` object as their argument and manage entering and leaving the Composite component.

The ConcreteVisitor class `FilenameQualifier` is derived from the `Visitor` class, and is responsible for returning the current qualified file name. The `FilenameQualifier` class overrides the `visitEnter()` and `visitLeave()` methods. The `FileListingVisitor` class is derived from the `FilenameQualifier` class and is used to refine the traversal behavior, *i.e.*, creating a hierarchical listing of qualified file names. The `FileSystemObject` class represents the Element component and implements the `accept()` and `getname()` methods that are overrided by the ConcreteElements `Directory` and `Leaf`.

```
1  class Visitor {
2  public:
3    virtual bool visitEnter (Directory *) = 0;
4    virtual bool visitLeave (Directory *) = 0;
5    virtual bool visit (File *) = 0;
6  };
7
8  class FileSystemObject {
9  public:
10   virtual string getName (void) const = 0;
11   virtual bool accept (Visitor *v) = 0;
12 };
13
14 class Directory : public FileSystemObject {
15 public:
16   Directory (const string & name)
17     : name_ (name) {  }
18
19   const string & getName (void) const { return this->name_;}
20   void add (FileSystemObject * child) { this->contents_.push_back (child); }
```


DOI: 10.1002/spe

```
21
22    bool accept (Visitor *v) {
23      // Try to enter the Directory.
24      if (v->visitEnter (this)) {
25        vector <FileSystemObject *>::iterator
26          it = this->contents_.begin(), it_end = this->contents_.end();
27
28        for (;it != it_end; ++it)
29          (*it)->accept (v);
30      }
31
32      // Notify the visitor we are leaving the Directory.
33      return v->visitLeave (this);
34    }
35
36  protected:
37    string name_;
38    vector <FileSystemObject *> contents_;
39  };
40
41  class File : public FileSystemObject {
42  public:
43    File (const string & name)
44    : name_ (name) {  }
45
46    const string & getName (void) const { return this->name_; }
47    bool accept (Visitor * v) { return v->visit (this); }
48
49  private:
50    string name_;
51  };
52
53  class FilenameQualifier : public Visitor {
54  public:
55    FilenameQualifier (void)
56      : path_ ("") { }
57
58    bool visitEnter (Directory * n) {
59      this->path_ += n->get_name() + "//";
60      return true;
61    }
62
63    bool visitLeave (Directory * n) {
64      path.resize (this->path_.size() - n->getName().size() - 1);
65      return true;
66    }
67
68    const string & getCurrentPath (void) const { return this->path_;}
69
70  protected:
71    string path_;
72  };
73
74  class FileListingVisitor : public FilenameQualifier {
75  public:
76    FileListingVisitor (void)
77    : level_ (0) {  }
78
79    bool visitEnter (Directory * n) {
80      FilenameQualifier::visitEnter (n);
81      this->println (n->getName ());
82
83      ++ this->level_;
84      return true;
85    }
86
87    bool visitLeave (Directory * n) {
```


DOI: 10.1002/spe

```
88      FilenameQualifier::visitLeave (n);
89
90      ++ this->level_;
91      return true;
92    }
93
94    bool visit (File * f) {
95      println (this->getCurrentPath () + f->getName ());
96      return true;
97    }
98
99    // Print the full qualified file name of the visited file.
100   void println (string s) {
101     for (int n = this->level_ * 4; n > 0; -- n)
102       cout << " ";
103     cout << s;
104   }
105
106 protected:
107   int level_;
108 };
109
110 //
111 // Main entry point of the application.
112 //
113 int main (int argc, char * argv []) {
114   Directory dir("root") ;
115   Directory temp("temp");
116   File f1("foo.txt");
117   File f2("bar.txt");
118
119   temp.add (&f1);
120   dir.add (&temp);
121   dir.add (&f2);
122
123   FileListingVisitor fv;
124   dir.accept (&fv);
125
126   return 0;
127 }
```

Listing 11: Source code snippet illustrating usage of Hierarchical Visitor pattern

Benefits. The following is a list of benefits for the Hierarchical Visitor pattern:

- The Hierarchical Visitor pattern provides enhanced navigational capability by allowing a programmer to track traversal depth (*i.e.*, hierarchical navigation) and short-circuit branch traversal (*i.e.*, conditional navigation).

Limitations. The following is a list of limitations for the Hierarchical Visitor pattern:

- The hierarchical navigation capabilities of this pattern can be used only for those hierarchies where a clear distinction can be made between the Composite and Leaf participants.

3.6. Generic Visitor Pattern

Problem. The original Visitor pattern requires that the return types of visit methods and the `accept()` methods to be known in advance. This becomes a drawback if the client later wants to use different return types for the visit and `accept()` methods.

Solution Approach. Generic programming techniques, such as templates in C++ and generics in Java and C#, can be used to address the drawbacks mentioned above. Dascalu et al. [27] implemented seven of the patterns proposed by the GoF—including the Visitor pattern—using C++ templates. The goal of their work was implementing design pattern automation [28].

Implementation. Lisiting 12 shows an example of how the `Element` and `Visitor` classes can use generic programming to alleviate the fixed return type problem of visiting methods (*i.e.*, the visit

DOI: 10.1002/spe

and accept method). As shown in this listing, for the `Visitor` base class, the `visitPerson()` method has a generic return type (25).

```
template <typename R>
class Object {
public:
  virtual void accept (Visitor <R> * v) = 0;
};

template <typename R>
class Person : public Object <R> {
public:
  Person (std::string name)
    : name_ (name) {}

  Person (int sal)
    : sal_ (sal) {}

  void accept (Visitor <R> * v) { std::cout << v->visitPerson (this)<<" : "; }

  std::string name_;
  int sal_;
};

template <typename R>
class Visitor {
public:
  virtual R visitPerson (Person <R> *p) = 0;
};

template <typename R>
class NameVisitor : public Visitor <R> {
public:
  R visitPerson (Person <R> *p) { return p->name_; }
};

template <typename R>
class SalaryVisitor : public Visitor <R> {
public:
  R visitPerson (Person <R> *p) { return  p->sal_;  }
};
```

Listing 12: C++ template code snippet illustrating usage of the Generic Visitor pattern.

Similar to C++ templates, in Java, generics can be used as shown in Listing 13.

```
public class NameVisitor <R> extends Visitor {
   public R visitPerson(Person <R> *p) { return p->name_; }
}
```

Listing 13: Java code snippet of the NameVisitor class.

In addition to the class-level template parameterization, both templates and generics allow method-level template parameterization (called *function templates* in C++ and *generic methods* in Java).

Benefits. The following is a list of benefits for the Generic Visitor pattern:

- The Generic Visitor pattern provides the developer with more flexibility when determining the type declarations of return types and arguments for the visit methods.
- The Generic Visitor pattern allows the client to change the visit method's return type with minimal impact.

Limitations. The following is a list of limitations for the Generic Visitor pattern:

- It can be hard to debug generic code when using C++ templates, which can make this pattern hard to implement for novice developers. However, debugging generics in Java and C# is comparatively easier.

DOI: 10.1002/spe

- When using templates, objects of a class with different type parameters are different types at run time; whereas, when using generics, type parameters are erased when compiled (*i.e., type erasure*).
- Unlike generics, the indiscriminate use of templates can lead to code bloat, resulting in excessively large executables.

An in-depth comparison of Java generics and C++ templates can be found in the article [29].

4. ALTERNATIVES TO THE ORIGINAL VISITOR PATTERN

This section discusses different alternatives to the original Visitor pattern to overcome some of the limitations introduced in Section 2. The alternatives discussed in this section either follow some or none of the structural aspects in the original Visitor pattern.

Table III. Summary of alternatives to the original Visitor pattern and the limitation each alternative addresses.

Pattern Name	Limitations Addressed	Section Discussed
Reflective Visitor Pattern	Prior knowledge of arguments and return type	4.1
	Adding new elements is difficult	
	Necessity of accept method	
Walkabout Class Visitor Pattern	Prior knowledge of arguments and return type	4.2
	Adding new elements is difficult	
	Necessity of accept method	
Dynamic Dispatcher Visitor Pattern	Adding new elements is difficult	4.3
	Necessity of accept method	
	Implementation inheritance not supported	
	Inability to access the special functions of ConcreteVisitor	
Case Class and Extractors in Scala	Necessity of accept method	4.4
	Partial visitation	
Generic Scala Visitor Library	Prior knowledge of the arguments and return type	4.5
	Little traversal control	
	Breaking encapsulation	

Table III gives a brief overview of different alternatives discussed throughout the remainder of this section. As shown in this table, different alternatives address different limitations of the original Visitor pattern (*i.e.*, not one approach addresses all the limitations). The remainder of this section discusses each of the alternatives presented in Table III in more detail using the following discussion format: the problem the alternative addresses, its solution approach, its implementation, and known benefits and limitations.

4.1. Reflective Visitor Pattern

Problem. The original Visitor pattern allows a designer to define new kinds of operations on an object structure without changing the classes of the objects [8]. The original Visitor pattern, however, exhibits added complexity when it comes to extending the object structure because it requires modifying, or rather updating, all the classes in the Visitor hierarchy.

Solution Approach. Mai et al. [8] addressed the limitation above by creating the *Reflective Visitor pattern* based on reflection techniques. The Reflective Visitor pattern allows the user to modify or update the object structure without the need to make any changes in the Visitor hierarchy. It

Prepared using speauth.cls
DOI: 10.1002/spe

accomplishes this by allowing the Visitor to perform runtime dispatching on itself when the client wants to visit a ConcreteElement.

Implementation. Figure 6 illustrates the Reflective Visitor pattern's structure. As shown in this

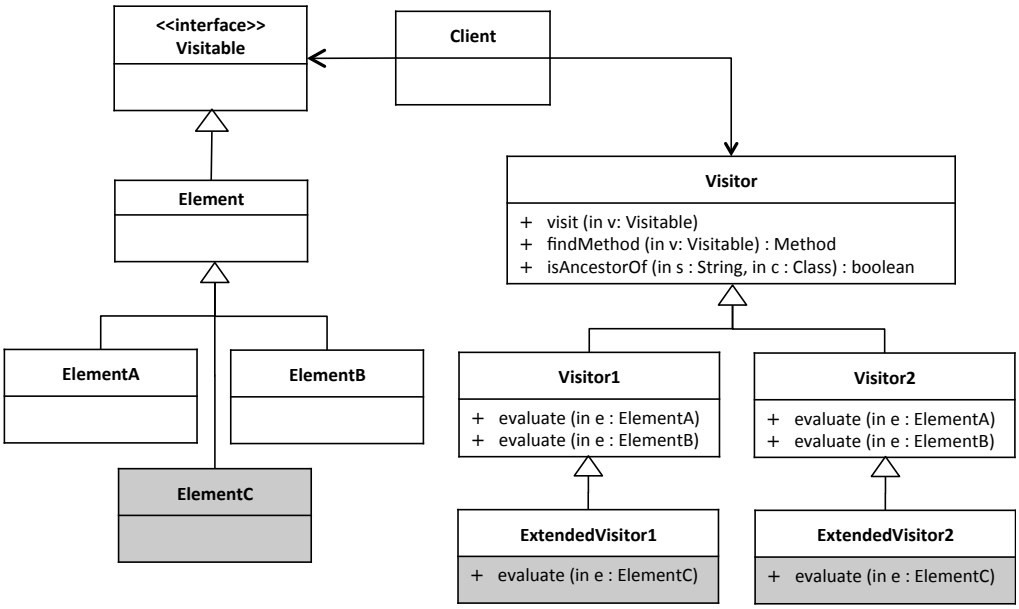

Figure 6. Structure of the Reflective Visitor pattern.

figure, the `Visitor` class defines a public `visit()` method. The client invokes this method to execute different operations on the object structure. It is then the responsibility of the `visit()` method to locate the concrete `evaluate()` operation in the Visitor hierarchy at runtime. Each `evaluate()` method has operations for its corresponding ConcreteElement participant. Finally, all the ConcreteElement participants that can be visited implement the `Visitable` interface, which provides runtime type information to the Visitor.

Benefits. The following is a list of benefits for the Reflective Visitor pattern:

- The Reflective Visitor pattern makes it easier to add new operations because it involves *simply* adding a new subclass to the Visitor hierarchy.
- The Reflective Visitor pattern makes it easier to add new ConcreteElements. This is because as the Visitor is responsible for dynamic dispatching, any operation acting on the newly added ConcreteElement can be defined in a new ConcreteVisitor participant.
- The runtime dispatching operation performed within the `Visitor` class eliminates cyclic dependencies between the object structure and Visitor hierarchy—thereby reducing its coupling.
- The reflection technique also eliminates the need for the `accept()` method, which in turn makes the source code more flexible and optimized. This is because removal of the `accept()` method eliminates an extra level of indirection.

Limitations. The following is a list of limitations for the Reflective Visitor pattern:

- The reflection technique reduces system efficiency. This performance penalty makes this pattern suitable only for non-critical systems.
- This pattern can be implemented only in programming languages that support reflection (*e.g.*, C# and Java). Otherwise, programming languages that do not support reflection, such as C++, can only leverage this pattern if third-party libraries exist that enable reflection [30, 31].
- The name of all the operations defined in the ConcreteVisitor participant should always be fixed. For example, in Figure 6 the name of all operations is fixed to `evaluate`. This is because the Visitor class uses reflection to locate the ConcreteElement's class information

DOI: 10.1002/spe

and invoke the appropriate `evaluate()` method that has the same argument as the ConcreteElement participant.

4.2. Walkabout Class Visitor Pattern

Problem. The implementation of the original Visitor pattern assumes that one knows the classes of all the objects to be visited beforehand. This makes it necessary to change the visitors whenever the class structure is changed. Moreover, the Visitor is dependent upon the `accept()` method for sending back the name of the class through dynamic binding.

Solution Approach. Palsberg et al. [10] addressed the limitations above by separating the concepts of "accessing sub-objects" and "actions taken on them". To access the structure of the objects, they use the Java class named *Walkabout*, which is the superclass for creating all visitors. The actions suitable to various kinds of objects are provided in the subclasses—thereby creating task specific visitors. The Walkabout class then uses reflection to determine the default access pattern for visitation. This information is then used to determine the internal structure of an object, including its fields.

```
1  class Walkabout {
2    void visit (Object v) {
3      if (v != null) {
4        // Use of reflection to visit Participants
5        if ([this has a public visit method for the class of v]) {
6          this.visit (v);
7        }
8        else {
9          if ([v is not of primitive type]) {
10           for ([each field f of v])
11             this.visit (v.f);
12         }
13       }
14     }
15   }
16 }
```

Listing 14: Pseudo-code of generic Walkabout class.

Implementation. As shown in Listing 14, the Walkabout class has just one method named `visit()`, which has a parameter of type `Object`. All visitors are then derived from the Walkabout class and implement `visit()` methods of interest without any further need for reflection.

Listing 15 shows a full Java code example (provided by Palsberg et al.) that uses the Walkabout class. As illustrated in the listing below, the indirection caused by the `accept()` method in the original Visitor pattern has been replaced by a direct recursive call. The direct recursive call now allows a ConcreteElement participant's visit method to be directly invoked without having to go through the `accept()` method unlike the original Visitor pattern.

```
1  interface List {}
2  class Nil implements List {}
3
4  class Cons implements List {
5     int head;
6     List tail;
7  }
8
9  class SumWalkabout extends Walkabout {
10   int sum = 0;
11   public void visit (Cons x) {
12     sum = sum + x.head;
13     this.visit (x.tail)
14   }
15 }
16
17 //
18 // Main entry point
19 //
```


DOI: 10.1002/spe

```
20  class MainProgram {
21    public static void main(String [ ] args) {
22      List l;
23      SumWalkabout sw = new SumWalkabout ();
24      sw.visit (l);
25      System.out.println (sw.sum);
26    }
27  }
```

Listing 15: Java source code snippet illustrating usage of the Walkabout class.

Benefits. The following is a list of benefits for the Walkabout Class Visitor pattern:

- The Walkabout class eliminates the need for the `accept()` method through the use of reflection.
- As the Walkabout class uses reflection, it is easy to add new elements to the element hierarchy—similar to the Reflective Visitor pattern presented in Section 4.1.

Limitations. The following is a list of limitations for the Walkabout Class Visitor pattern:

- The reflection technique has performance overhead because reflection involves types that are dynamically resolved. Moreover, code using reflection is generally slower than the equivalent native code [10].
- Palsberg et al. implemented the Walkabout Class Visitor pattern using a language that requires reflection support. It therefore has the same limitations as the Reflective Visitor pattern with respect to reflection (see Section 4.1).

4.2.1. Walkabout Class Visitor Pattern for Fixed Object Structures. The use of reflection in the Walkabout Class Visitor pattern poses a significant performance penalty. Palsberg et al. improved the Walkabout Class Visitor pattern by using type casts and *instanceof* operator for scenarios where the ConcreteElement participants, and its fields, are fixed.

Listing 16 shows the walkabout class for fixed object structures where the ConcreteElement participants present in the object structure are `Book` (fields: `getName` and `getBookId`), `Article` (fields: `getAuthor` and `getArticleId`) and `Patron` (fields: `getName` and `getPatronId`). In this example, the type casts and *instanceof* operators are used to directly access the fields of a class. It is therefore no longer necessary to specify the visit method for the ConcreteElement of the current object [10].

```
1   class Walkabout {
2     void visit (Object v) {
3       if (v != null) {
4         // use of Reflection technique
5         if (this has a public visit method for the class of v) {
6           this.visit (v);
7         }
8         else if (v instance of Book) {
9           this.visit(((Book) v).getName ());
10          this.visit(((Book) v).getBookId ());
11        }
12        else if (v instance of Article) {
13          this.visit(((Article) v).getAuthor ());
14          this.visit(((Article) v).getArticleId ());
15        }
16        else if (v instance of Patron) {
17          this.visit(((Patron) v).getName ());
18          this.visit(((Patron) v).getPatronId ());
19        }
20      }
21    }
22  }
```

Listing 16: Walkabout for Fixed Classes.

DOI: 10.1002/spe

As a result, the code is more robust and does not use reflection as much when compared to the original Walkabout Class Visitor pattern (see Section 4.2). On the other hand, the modification still uses reflection, which limits it in the same ways as the original Walkabout Class Visitor pattern and the Reflective Visitor pattern (see Section 4.1). Additionally, the modified Walkabout class requires the object structure to be fixed and known *a priori* to the client, which makes the object structure very difficult to extend.

4.2.2. Modified Walkabout Class Visitor Pattern using Accept Method. The Walkabout Class Visitor pattern for fixed object structures still uses some degree of reflection, which impacts performance. Palsberg et al. [10] addressed this limitation by using the `accept()` methods on each of the ConcreteElement participants. The `accept()` methods are then used to invoke the corresponding visit method for the ConcreteElement participant. Dynamic binding in the `accept()` methods ensures that the Visitor knows the classes of objects to be visited, thereby allowing the Walkabout class to be used as an interface and create task-specific visitors.

This modification, however, makes the new Walkabout class conform to the structure of the original Visitor pattern. The Modified Walkabout Class Visitor Pattern using Accept Method therefore does not have any benefits over the original Visitor pattern introduced in Section 2.

4.3. Dynamic Dispatcher Visitor Pattern

Problem. The intrusive nature of the original Visitor pattern makes it unsuitable for extending frameworks and libraries. This is because of two reasons: firstly, the presence of `accept()` method in the object structure, and secondly, the existence of cyclic dependencies between classes and subclasses (*i.e.*, the Visitor knows all the domain classes through the parameters of the visit method and vice versa). Moreover, the Visitor pattern does not support implementation inheritance (*i.e.*, the pattern is unable to leverage inherited operations in the ConcreteVisitor participant). This makes it necessary to change all the other ConcreteVisitor participants if one ConcreteVisitor participant requires a special method for a ConcreteElement not included in the `Visitor` participant. Finally, manually simulating implementation inheritance is not only time consuming, but also error prone and *hard* to maintain.

Solution Approach. To overcome the limitations discussed above, Buttner et al. [15] designed an alternative to the original Visitor pattern called the *Dynamic Dispatcher Visitor* pattern. This pattern has the ability to dynamically choose the most appropriate visit method for a particular ConcreteElement.

Implementation. The general structure of the Dynamic Dispatcher Visitor pattern is shown in Figure 7. As shown in this figure, the `DispatcherFactory` and `Dispatcher` together form the dispatching framework. Visitors are arbitrary classes that have implemented visit methods of interest. The `DispatcherImpl` object, which is created by the `DispatcherFactory`, has a reference to its visitor. Finally, the `dispatch()` method selects the visit method that has an argument that is most closely related to the ConcreteElement being visited. If there is ambiguity when selecting the most appropriate visit method, then the dispatcher fails. Buttner et al. suggest that ambiguity can be avoided in Java and C# by restricting the allowed argument types in the visit methods to be ConcreteElement types [15]. For languages like C++, however, this ambiguity persists.

Benefits. The following is a list of benefits for the Dynamic Dispatcher Visitor pattern:

- The Dynamic Dispatcher Visitor pattern can be used to extend frameworks and libraries because it overcomes the `accept()` method and cyclic dependencies between domain classes limitations.
- The Dynamic Dispatcher Visitor pattern facilitates implementation inheritance in visitor operations. This is because specialized visitor operations can be added to individual domain classes similar to overriding methods in the domain class hierarchy [15]. The Dynamic Dispatcher Visitor pattern therefore makes the code more robust against future changes.
- The absence of dispatching code supports clear and concise expression of operations on the object structure.

Prepared using speauth.cls
DOI: 10.1002/spe

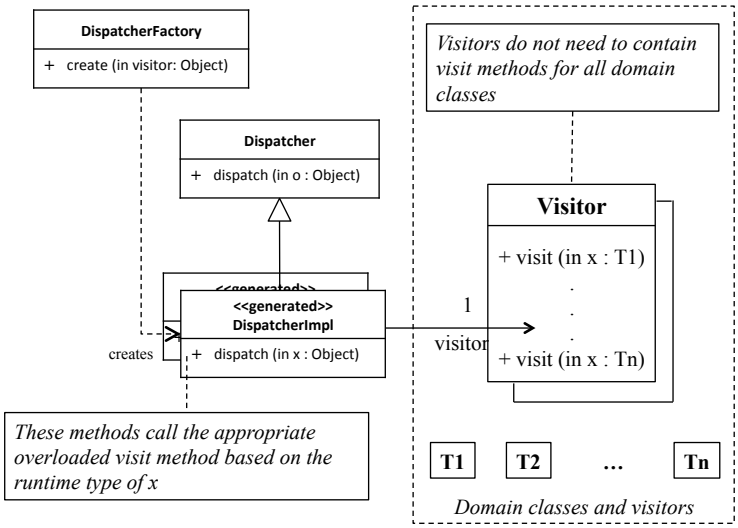

Figure 7. Structure of the Dynamic Dispatcher Visitor pattern.

Limitations. The following is a list of limitations for the Dynamic Dispatcher Visitor pattern:

- The additional dispatch code for calling the appropriate `visit()` method decreases performance.
- Type errors can occur at runtime because the dispatcher performs type checking at runtime. If an appropriate visit method is not located, then an exception is thrown.

4.4. Case Class and Extractors in Scala

Problem. The original Visitor pattern allows object-oriented pattern matching by separating the object structure from the operations. The Visitor pattern, however, causes a high degree of notational overhead for framework constructions [32]. This is because the Visitor class has visit methods for each ConcreteElement participant and each of the ConcreteElement participants implements an `accept()` method. Moreover, Visitors are not easily extensible, *i.e.*, neither new patterns nor new alternatives can be created without completely redesigning the Visitor framework.

Solution Approach. Emir et al. [32] provided a novel solution to the object-oriented pattern matching problem by using a combination of case classes and extractors in the Scala programming language. Case classes in Scala are normal classes with a proceeding *case* modifier. The special features of case classes are:

- They allow pattern matching on their constructor;
- They allow a user to construct instances of these classes without using the *new* keyword;
- The constructor arguments are accessible from outside using automatically generated accessor functions;
- The `toString()` method is automatically defined to print the name of the case class and all its arguments;
- The `equals()` method is automatically defined to compare two instances of the same case class structurally rather than by identity;
- The `hashCode()` method is automatically redefined to use the hashCodes of constructor arguments.

Extractors provide another way to define a pattern without a case class. It has two major components: *injection*, a method that takes an argument and yields an element of a given type; and *extraction*, a method that extracts parts of the given type. Pattern matching using case classes does not require any notational overhead for the class hierarchy. Case classes, however, expose object representations; and they also make it easy to add new alternatives to the Visitors. It, however, is

DOI: 10.1002/spe

not possible to add new patterns using case classes because patterns have one-to-one relationships with the ”types” of case classes. This shortcoming is eliminated by pairing case classes with extractors [32].

Implementation. Listing 17 shows a class hierarchy expressed in Scala programming language. As shown in this listing, the base class `Expr` represents expressions, the `Var` class represents variables, the `Num` class represents numeric literals, and the `Mul` class represents multiplication operation. Line 8 shows an example of a simplification rule that requires pattern matching. This rule states that if a number x is multiplied by 1, it returns the same number x.

```
1  // Class hierarchy
2  class Expr
3  class Num (val value : int) extends Expr
4  class Var (val name : String) extends Expr
5  class Mul (val left : Expr, val right : Expr) extends Expr
6
7  // Simplification rule
8  new Mul (x, new Num(1)) ⟹ x
```

Listing 17: Expression class hierarchy.

Listing 18 shows the class hierarchy and the simplification rule expressed using case classes. Lines 14–26 show a completely expanded `Mul` case class, and lines 29–32 show the `Mul` class represented using an extractor.

```
1  // Class hierarchy
2  trait Expr
3  case class Num (value : int) extends Expr
4  case class Var (name : String) extends Expr
5  case class Mul (left : Expr, right : Expr) extends Expr
6
7  // Simplification rule
8  e match {
9    case Mul(x, Num(1)) ⟹ x
10   case _ ⟹ e
11 }
12
13 // Expanded Mul case class
14 class Mul (_left : Expr, _right : Expr) extends Expr {
15   //Accessors for constructor arguments
16   def left = _left
17   def right = _right
18
19   //Standard methods
20   override def equals (other : Any) = other match {
21     case m : Mul ⟹ left.equals (m.left) && right.equals (m.right)
22     case_ ⟹ false
23   }
24   override def hashCode = hash (this.getClass, left.hashCode, right.hashCode)
25   override def toString = "Mul(" + left+","+right+")"
26 }
27
28 // Representing Mul class using extractors
29 object Mul {
30   def apply (left : Expr, right : Expr) = new Mul (left, right)
31   def unapply (m : Mul) = Some (m.left, m.right)
32 }
```

Listing 18: Source code snippet illustrating use of case classes and extractors.

Benefits. The following is a list of benefits of using case class and extractors in Scala:

- The use of case classes makes the program more generic. The developer therefore is not required to have prior knowledge of arguments and return types of visiting methods.
- The use of this alternative makes it easier to extend the object structure as well as the visitor hierarchy.

DOI: 10.1002/spe

Limitations. The following is a list of limitations of using case class and extractors in Scala:

- Case classes cannot be inherited. A common design choice therefore is to build up a class hierarchy of traits using case classes as the leaf nodes of the tree.

4.5. Generic Scala Visitor Library

Problem. The original Visitor pattern requires developers to make important design decisions (*e.g.*, return type of visiting methods, shape of the element hierarchy, and location of traversal logic) during early phases of the design process. Moreover, the original Visitor pattern also prevents capturing certain cross-cutting concerns, such as execution tracing and memoization of results [33].

Solution Approach. Oliveira et al. [33] represented the Visitor pattern as a reusable, generic, type-safe component by using type system features of modern object-oriented languages like Scala. They designed a *Scala Visitor Library*, which overcame the limitations of the original Visitor pattern mentioned above. The Scala Visitor Library allows implementation choices specified in Table IV to occur later in the design process. Moreover, Oliveira et al. also provided semantics for a generalized *algebraic datatype notation* using visitors built with the Scala Visitor Library.

Table IV. The implementation choices a user has to make early in the design process.

Implementation Decision	**Options**
Return type of visit and accept methods	Imperative Visitor (*i.e.*, the methods return `void`) or Functional Visitor (*i.e.*, the methods return values)
Location of traversal code	Internal Visitor (*i.e.*, traversal code is in the accept methods) or External Visitor (*i.e.*, traversal code is in the visit methods)
Shape of object structure being traversed	N/A
Cross-cutting concerns	Tracing of execution, memoization of results, etc.

Implementation. Table V shows the correspondence between concepts of the original Visitor pattern (refer Section 2) and the Scala Visitor Library. Likewise, Listing 19 provides an example of

Table V. Corresponding concepts between Scala Visitor Library and the original Visitor pattern.

Scala Visitor Library	**Original Visitor Pattern**
data T	Element
constructor	ConcreteElement
(D)$Case_T$	Visitor
V **extends** (D)$Case_T$	ConcreteVisitor
new (D)$Case_T$	Anonymous ConcreteVisitor

the tree structure represented in the new datatype notation [33].

```
1  // Tree definition in the new data notation
2  data Tree {
3    constructor Empty
4    constructor Fork (x : int, l : Tree, r : Tree)
5  }
6
7  // Anonymous ConcreteVisitor depth_1
8  def depth_1 = new CaseTree [External, int] {
9     def Empty = 0
10    def Fork (x : int, l : R[TreeVisitor], r : R[TreeVisitor]) =
11        1 + max (l.accept (this), r.accept (this))
12 }
13
```


DOI: 10.1002/spe

```
14  // Anonymous ConcreteVisitor depth2
15  def depth2 = new CaseTree [Internal, int] {
16      def Empty = 0
17      def Fork (x : int, l : R[TreeVisitor], r : R[TreeVisitor]) =
18          1 + max (l.accept (this), r.accept (this))
19  }
```

Listing 19: Source code snippet illustrating the tree structure represented in the new datatype notation.

The various modifications over the original Visitor pattern depicted in Listing 19 are as follows:

- **Traversal strategy.** The visitors have a choice of the following traversal strategies: internal and external. *Internal traversal* is when the traversal code is in the accept methods, and *external traversal* is when the traversal code is in the visit methods. As shown on line 8 and line 15, the traversal strategy is parameterized on the ConcreteVisitor participant instead of being fixed on the Visitor component.
- **Functional notation.** Scala allows functions to be represented as objects. This, in turn, allows developers to leverage functional notation and make visitors a subclass of functions with composites as arguments. For example, line 3 in Listing 20 shows how the ConcreteVisitor $depth_2$ makes a recursive call to itself instead of repeatedly calling the accept method.
- **Advice and modular concerns.** The Scala Visitor Library provides more traversal control, which allows non-functional concerns (*i.e., advice*) to be decoupled from base programs and placed into localized modules. The Scala Visitor Library includes several commonly used pieces of advice, which is shown in Table VI. The library also provides templates extending the advice listed in Table VI with user-defined advice [33].
- **Multiple dispatching.** The Scala Visitor Library allows developers to simulate multiple dispatching by using nested external visitors.

```
1  def depth2 = new CaseTree [Internal, int] {
2      def Empty = 0
3      def Fork (x : int, l : Tree, r : Tree) = 1 + max (depth2(l), depth2(r))
4  }
```

Listing 20: Source code snippet illustrating functional notation with the Scala Visitor Library.

Table VI. Commonly used *advices* in the Scala Visitor Library

Advice	**Description**
Basic	Simple dispatcher that defines the default behavior of a visitor.
Memo	Memoization of results.
Advice	Template for defining new dispatchers that have *before* and *after* methods, which are triggered before and after calls.
Trace	Tracing a computation by printing the input and output, which is implemented using *Advice* as template

Benefits. The following is a list of benefits of using the Generic Scala Visitor Library:

- The Generic Scala Visitor Library allows the developer to make important design and implementation decisions later on in the development process.
- The Visitor design pattern has been represented as a generic, type-safe visitor software component [34]. Its type-safety is guaranteed by the modular type system, which is an advantage over designs that make use of reflection.

Limitations. The following is a list of limitations of using the Generic Scala Visitor Library:

- A major disadvantage of using case classes is that they cannot be inherited. A common design choice therefore is to build up a class hierarchy of traits, using case classes as the leaf nodes of the tree.

DOI: 10.1002/spe

- Another limitation of this library is that it relies on advanced type system features that are available in Scala and not in other languages, such as C++.

4.5.1. Advanced Solutions. The original Visitor pattern lacks the ability to solve the expression families problem (*i.e.,* achieving reusability and composability across the components involved in a family of related datatypes and corresponding operations over those datatypes) [35]. The original Visitor pattern fails to address this problem because it does not preserve modular and static type-safety (*i.e.,* no modification, duplication, re-compilation or re-type-checking of the existing code is required) when new components are added. The Visitor pattern also fails to provide a high degree of composability and decoupling of components. Oliveira provided a solution to this problem in Scala using modular visitor components. There are two variations of the modular visitors: *modular internal visitor* and *modular external visitor* (inspired by two datatype encodings in the lambda calculus i.e. Church encoding [36] and Parigot encoding [37] respectively), that can be alternatively used to solve the expression families problem. As stated by Oliveira, the Generic Scala Visitor Library can be used to make this design choice later in the development process [35].

5. SUMMARY OF EXTENSIONS AND ALTERNATIVES

This section summarizes the various extensions and alternatives discussed in Section 3 and Section 4. It also presents the scenarios under which a user should select a particular extension or alternative over the original Visitor pattern and vice versa.

- **Extended Types Visitor Pattern.** The Extended Types Visitor Pattern is useful when a design contains separate element hierarchies where each hierarchy is captured by its own visitor, such as base types and extended types. The dynamic type casting allows us to invoke the visit methods on the extended types instead of on the base types. The developer, however, must be aware of the ConcreteVisitor participant that will be used to typecast the Visitor participant in the `accept()` method.
- **Acyclic Visitor Pattern.** The Acyclic Visitor pattern is useful in the following scenarios [23]:
  - There are operations that need to be applied only on certain elements in the Element hierarchy;
  - The Element hierarchy is frequently extended with new elements; or
  - The recompilation, relinking, retesting or redistribution of `ConcreteElement` participants is expensive.
- **Visitor Combinators Visitor Pattern.** The Visitor Combinators Visitor pattern is beneficial for designs that require complete tree traversal control. This is because *visitor combinators* capture basic functionality related to tree traversal control, and can be combined to create complex visitors that provide new functionality to enhance tree traversal control. Lastly, this pattern should be used when the design requires ordering application of the ConcreteVisitor participants, or when one ConcreteVisitor participant requires the output of another ConcreteVisitor participant as input.
- **Normal Form Visitor Pattern.** The use of Normal Form Visitor pattern improves the maintainability and extensibility of the system. This is because the Normal Form Visitor pattern conforms the original Visitor pattern design to the SOLID principles [19] of object-oriented design. The goal of the SOLID principles is to have ease of maintenance and improved extensibility. It is worth noting that the Normal Form Visitor pattern may, however, complicate the overall design.
- **Hierarchical Visitor Pattern.** The Hierarchical Visitor pattern is a good choice in the following scenarios:
  - The Element hierarchy uses the Composite pattern [3];
  - The visitor has to determine if one `Composite` is derived from another `Composite` or if they are siblings; or

DOI: 10.1002/spe

  - Tree traversal requires skipping branches based on conditions.

  This is because the Hierarchical Visitor pattern allows developers to have navigational capabilities on hierarchical object structures.
- **Basic Generic Visitor Pattern.** The Basic Generic Visitor Pattern is a good choice when the user requires flexibility in terms of the return types of visiting methods (*i.e.* `visit` and `accept` methods). However, the users have to make a clear choice between C++ templates and Java/C# generics based on their advantages and disadvantages.
- **Reflective Visitor Pattern.** The Reflective Visitor pattern is an excellent choice over the original Visitor pattern in the following situations [8]:
  - The run-time efficiency is not a major concern in the design;
  - Distinct and unrelated operations are performed on the `ConcreteElement` participants;
  - The Element structure is changed often to fit new requirements;
  - The Element structure and Visitor hierarchy needs to be reused in the future; or
  - The developer wants to hide implementation details from the client and provide them with a unified stable operation interface.

  The Reflective Visitor pattern makes it easier to add new ConcreteVisitor and ConcreteElement participants through the use of reflection. It also reduces the coupling between the element structure and the Visitor Hierarchy by eliminating the need for the `accept()` method. This pattern, however, can be implemented only in languages that support reflection such as Java and Smalltalk.
- **Walkabout Class Visitor Pattern.** The Walkabout Class Visitor pattern also uses reflection, and has the same benefits and limitations as the Reflective Visitor pattern. The Walkabout Class Visitor pattern, however, is a better choice than Reflective Visitor pattern for fixed object structures (*i.e.*, the Element hierarchy does not change and is known to the client).
- **Dynamic Dispatcher Visitor Pattern.** The Dynamic Dispatcher Visitor is a good choice when clients want to extend frameworks and libraries. It is also a good choice when clients want to have only some of the visit methods in the visitors, *i.e.*, the ConcreteVisitor participant is not interested in visiting all the ConcreteElement participants.
- **Generic Scala Visitor Library.** The Generic Scala Visitor Library is a good choice when the user wants to make important implementation decisions (see Table IV) later in the development process. The Generic Scala Visitor Library can also ensure type-safety. Oliveira depicted the use of two alternative variations of modular visitors: *internal modular visitor* and *external modular visitor* to solve the `Expression Family Problem` [35]. The use of Generic Scala Visitor Library helps in solving EFP by providing the user a choice between the two visitors.
- **Case classes and extractors.** The use of Scala case classes and extractors helps overcome the *breaking encapsulation* problem associated with the original Visitor pattern. The developer, however, is restricted to representing composite nodes as traits and leaf nodes as case classes. This is because case classes cannot be inherited.

6. CONCLUDING REMARKS

The Visitor pattern is one of the Gang of Four's software design patterns that separates an algorithm from the object structure it operates on. Since the introduction of the Visitor pattern in 1995, numerous researchers have proposed several modifications to the original Visitor pattern. These modifications were intended to overcome certain limitations of the original Visitor pattern. This article highlights those major enhancements and categorizes them into two groups based on their degree of variation in maintaining the structural integrity of original Visitor pattern: *extended* and *alternatives*.

Based on our survey and classification, one can see that not one modification addresses all the limitations of the original Visitor pattern. Instead, software system developers must rely on several

DOI: 10.1002/spe

different approaches to overcome all the limitations—some of which are language specific, such as Reflective Visitor and Walkabout Class Visitor. As there is no single modification to address all the limitations of the original Visitor pattern, it is the responsibility of the software system developer to determine if addressing a limitation in the original Visitor pattern is a cost-effective solution in the long-run. We believe, however, this article will help those who are faced with this decision because it provides a comprehensive reference to different extensions and alternatives.

REFERENCES

1. Sandhu PS, Singh PP, Verma AK. Evaluating Quality of Software Systems by Design Patterns Detection. *Proc. Int. Conf. Advanced Computer Theory and Engineering ICACTE '08*, 2008; 3–7, doi:10.1109/ICACTE.2008.205.
2. Khaer MA, Hashem M, Masud MR. On Use of Design Patterns in Empirical Assessment of Software Design Quality. *Proc. Int. Conf. Computer and Communication Engineering ICCCE 2008*, 2008; 133–137, doi:10.1109/ICCCE.2008.4580582.
3. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
4. Tichy WF. A Catalogue of General-Purpose Software Design Patterns. *Proc. Technology of Object-Oriented Languages and Systems TOOLS 23*, 1997; 330–339, doi:10.1109/TOOLS.1997.654742.
5. Tao K, Palsberg J. *The Java Tree Builder*. Purdue University 1997.
6. Metamata, Microsystems S. JavaCC 2000. URL `www.suntest.com/JavaCC/`.
7. Ledeczi A, Maroti M, Bakay A, Karsai G, Garrett J, Thomason C, Nordstrom G, Sprinkle J, Volgyesi P. The Generic Modeling Environment 2001.
8. Mai Y, de Champlain M. Reflective Visitor Pattern. *6th Annual European Conference of Pattern Languages of Programs (EuroPLoP 2001)*, 2001.
9. Visser J. Visitor Combination and Traversal Control. *16th ACM SIGPLAN conference on OOPSLA*, NY, USA, 2001; 270–282.
10. Palsberg J, Jay CB. The Essence of the Visitor Pattern. *COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, 1998; 9–15.
11. Mai Y, de Champlain M. A Pattern Language to Visitors. *8th Annual Conference of Pattern Languages of Programs, PLoP 2001*, 2001.
12. Lippman S. *Inside the C++ Object Model*. Addison-Wesley, 1996.
13. Martin RC. The Open/Closed Principle. *C++ Report* January 1996; .
14. Bravenboer M, Visser E. Guiding Visitors: Separating Navigation from Computation. *Technical Report*, Institute of Information and Computing Sciences, Utrecht University, Utrecht, Netherlands 2001.
15. Büttner F, Radfelder O, Lindow A, Gogolla M. Digging into the Visitor Pattern. *Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE2004)*, Banff, Alberta, Canada, 2004; 135–141.
16. Martin RC. The Dependency Inversion principle. *C++ Report* May 1996; .
17. Martin RC. The Interface Segregation principle. *C++ Report* August 1996; .
18. Xiao-Peng Z, Yuan-Wei B. Improved Visitor Pattern Based on Normal Form. *Proc. Second Int Education Technology and Computer Science (ETCS) Workshop*, vol. 3, 2010; 434–437, doi:10.1109/ETCS.2010.284.
19. Martin RC. Design Principles and Design Patterns. *Object Mentor* 2000; :1–34.
20. Nordberg III M. The Variations on the Visitor Pattern. PLoP 96 Writers Workshop September 1996.
21. Vlissides J. Visitor in Frameworks. *C++ Report* 1999; **11**(10):40–46.
22. ISO/IEC. International Standard: Programming Languages - C++. Number 14882:1998(E) in ASC X3 September 1998.
23. Martin RC. Acyclic Visitor. *Notes* 1995; :93103.
24. van Deursen A, Visser J. Building Program Understanding Tools Using Visitor Combinators. *Proc. 10th Int Program Comprehension Workshop*, 2002; 137–146, doi:10.1109/WPC.2002.1021335.
25. Beeri C, Bernstein PA, Goodman N. A Sophisticate's Introduction to Database Normalization Theory. *4th International Conference on Very Large Data Bases*, West Berlin, Germany, 1978; 113–124.
26. Falco RD. Hierarchical Visitor Pattern. `http://c2.com/cgi/wiki?HierarchicalVisitorPattern`.
27. Dascalu S, Hao N, Debnath N. Design Patterns Automation with Template Library. *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology 2005* 2005; :699–705.
28. Bulka A. Design Pattern Automation. *Proceedings of the 3rd Asia-Pacific Conferenceon Pattern Languages of Programs*, 2002.
29. Ghosh D. Generics in Java and C++: a comparative model. *SIGPLAN Not.* May 2004; **39**(5):40–47.
30. Devadithya T, Chiu K, Lu W. C++ Reflection for High Performance Problem Solving Environments. *SpringSim '07: Proceedings of the 2007 Spring Simulation Multiconference*, Society for Computer Simulation International: San Diego, CA, USA, 2007; 435–440.
31. Leitner A, Eugster P, Oriol M, Ciupa I. Reflecting on an Existing Programming Language. *TOOLS EUROPE 2007 - Objects,Models, Components, Patterns*, 2007.
32. Emir B, Odersky M, Williams J. Matching Objects with Patterns. *Computer* 2007; **4609**(1):273–298.
33. Oliveira B C d S and Wang M and Gibbons J. The Visitor Pattern as a Reusable, Generic, Type-safe Component. *ACM Sigplan Notices* 2008; **43**(10):439–456.
34. Oliveira B C d S . Genericity, Extensibility and Type-safety in the Visitor Pattern. PhD Thesis, University of Oxford 2007.
35. Oliveira B C d S. *Modular Visitor Components*, vol. 5653. Springer, 2009.

DOI: 10.1002/spe

36. Böhm C, Berarducci A. Automatic synthesis of typed -programs on term algebras. *Theoretical Computer Science* 1985; **39**:135 – 154.
37. Parigot M. Recursive programming with proofs. *Theor. Comput. Sci.* Mar 1992; **94**(2):335–356.

DOI: 10.1002/spe