Techniques for traversal operation on an object structure: a comparison

Jurij Mihelič, Igor Rožanc
Faculty of Computer and Information Science
University of Ljubljana
Večna pot 113, 1000 Ljubljana, Slovenia
{jurij.mihelic, igor.rozanc}@fri.uni-lj.si

Abstract. *In the paper we present several approaches of a traversal operation implementation on an object structure. Such operations and structures are often used in engineering software, e.g., in compiler design, code analytics, plagiarism-detection tools, etc. We focus on the following approaches (and their variants): dedicated methods, type checking and casting, visitor design pattern, and reflection based approach. The main goal of this paper is to explore and compare those approaches on a clear working example. Additionally, we describe several variants of reflection-based approach rather than the most straightforward one usually implemented. The presented approaches vary in many properties such as programming skills involved, separation of operation from the structure, performance, extensibility etc. Our detailed presentation enables an effective comparison based on such criteria which present the technical and practical aspects of each approach. In this way programming practitioners gain a complete insight into pros and cons of different traversal implementations.*

Keywords. traversal, object structure, visitor, design pattern, type casting, reflection, comparison

1 Introduction

In practice software engineering contains study of many different areas on how to efficiently solve actual problems using software. The definition of how the software is produced is one important aspect of these studies, which are aimed on research of new programming techniques and approaches [1].

The successful design of object-oriented program is hard. Even experienced designers have problems to find a general enough solution, which is flexible and reusable. Such solution enables the designers to find recurring patterns of classes and communicating objects which are common in many object-oriented systems. These design patterns are reused to solve specific problems and make the object-oriented design flexible [2].

A good example of design pattern is the visitor pattern. It enables the definition of a traversal operation performed on all elements of object structure without changing the classes of the elements on which it operates [2]. Obviously, its use enables developer to write a simpler and clearer solution, even in the case when the access to element classes is unavailable. In the literature there are several examples of visitor design pattern use [3], especially in the area of compilers, for instance SIC/XE assembler [4].

However the use of traditional visitor has an important demand - it has to know the classes of all objects it visits in advance. When the structure changes, the visitor has to be rewritten. Additionally, each class must have an accept method, which is used to pass the name of the class back to the visitor through dynamic binding [2, 5, 6].

This can be omitted with the use of reflection to access the structure of the objects it visits, thus the access to objects is separated from acting on them [7, 8]. Unfortunately, this totally flexible solution seems to be considerably slower [6].

In general, the traversal of all elements can be implemented in several ways. Simple implementations are designed ad-hoc and without the use of design patterns. They are (partly) implemented in each class of the elements visited, usually by adding similar parts of code to each. Better solutions introduce less or even none interference with those classes, but they are more complex to design and use. While sometimes the latter implementations are preferable or even the only possible (i.e., when classes of elements visited are unchangeable), in other cases such implementation may be over-designed as they introduce unnecessary complexity, thus yielding problems of solution understanding, development and maintenance. Consequently the decision on most suitable implementation is problem specific. However to be able to select the optimal visitor implementation the program designer needs a clear understanding of different approaches.

The goal of this paper is to address this issues. In particular, we examine the whole range of traversal implementations in detail and compare them using different criteria. To clearly introduce specifics we present

a working example of all implementations and discuss their characteristics. The comparison in last part of article enables practitioners to access the suitability of the solutions presented for their specific needs.

In Section 2 we present necessary preliminaries concerning the definition of our working example. The Section 3 introduces the simplest traversal implementation by dedicated methods. The Section 4 presents the implementation via run-type checking and type casing in two ways: as a structured approach and a proper object-oriented approach. The traditional visitor defined as a design pattern is demonstrated in two ways in Section 5. Section 6 presents the visitor implemented by reflection. In Section 7 the comparison of all approaches is presented. The last section contains the conclusion and future work.

2 Preliminaries

The aim of all techniques presented is to traverse an object structure and to apply some sort of processing on every visited object. To demonstrate the techniques we use an example object structure. The motivation for the example comes from the field of compilers and programming languages. The structure represents an abstract syntax tree (AST) of a hypothetical programming language. AST is a common approach to represent a parsed source code. It is usually processed and/or transformed in multiple passes to finally obtain a compiled code [9]. AST consists of nodes (representing programming constructs) and links (also called edges) from nodes to sub-nodes.

The example AST supports only a few programming constructs, but enough to demonstrate the traversal process. Each construct is represented by its own class. The complete class hierarchy is as follows. The base interface for the objects to be visited is the Visitable interface:

```
interface Visitable {}
```

Depending on the traversal technique it may be filled with various details (explained later in the text).

The base class representing AST nodes is Node. It is an abstract class, thus it can not be instantiated. Several other classes such as Comment, Print, PrintBold and Block represent various programming constructs and statements, which are also descendants of an abstract class Statement. The class hierarchy is the following (the subtyping relation is represented with line indentation):

```
interface Visitable
   abstract class Node
      class Comment
      abstract class Statement
         class Print
            class PrintBold
         class Block
```

This simple hierarchy exhibits important relations between nodes. It includes nodes related by inheritance, e.g., Print and PrintBold. Additionally, it includes a composite node Block which can, without loss of generality, contain two nodes.

We define several example operations on the structures represented by this class hierarchy:

print This kind of traversal pretty prints the program represented by a given AST.

dump Similar to the print operation, but the classes of given AST nodes are printed.

exec Executes the program represented by a given AST. In our case comments are ignored, Print's do the obvious, and Block delegate execution to its composite elements.

size Calculates and returns the size of a given AST. This operation demonstrates how to return a result of operation to the caller.

compile Compiles the program represented by a given AST. In our example, the compilation is only a simple transformation of a given AST, where comments are removed and PrintBold is replaced with a corresponding combination of Print's and Block's.

The above class hierarchy, object structure and traversal operations are used in the following sections as a demonstration example. The complete source code for all the presented examples can be obtained online [10] from the GitHub portal.

3 Dedicated methods approach

The approach described in this section is the most straightforward one. The main idea is to implement each operation on an object structure with a set of methods. In particular, for each class in the object-structure class hierarchy (i.e., instantiable class extending the Visitable interface) a method implementing a particular visit operation must be provided.

To accomplish this one must first provide the interface which declares all the methods – one for each operation. For example, in our case of five different operations on the object structure the Visitable interface looks like this:

```
interface Visitable {
   void print(int indent);
   void dump();
   void exec();
   int size();
   Node compile();
}
```

Notice that, two of the methods, namely int size() and Node compile() also return the result of the operation.

Afterwards, for each class the programmer has to implement these methods. Let us here present the code

excerpt for the Block class (only methods exec() and size()):

```
class Block extends Statement {
   void exec() {
      first.exec();
      second.exec();
   }

   int size() {
      return first.size() + second.size();
   }
   ...
}
```

For example, the execution of a block, i.e., the Block.exec() method, sequentially delegates the execution to both parts composing the block. Similarly, the size-calculation operation, i.e., the size() method, simply sums the sizes of both block parts (assuming no additional cost for the block size).

The advantage of this approach is in its simplicity and object orientation. However, in the case of a complex object structure consisting of many classes a lot of similar methods must be implemented for each operation. These methods are distributed all over the code implementing the class hierarchy of object structure. As a result the hierarchy implementation is polluted with programming code implementing operations, i.e., the object structure implementation is not *clean*. Consequently, the full separation of the object structure and operation implementations is not possible with this approach.

On the other side, due to the simplicity of the dedicated methods approach it may be suitable for smaller object structures without many operations involved, where a complete control over source code is at hand.

4 Type checking and type casting

In this section we present two variants of an approach that is based on the programming language run-time type checking capability and type casting functionality mainly. In Java the former is presented via the instanceof built-in operator, and the latter via well-known "$(type)\ object$" syntax which converts a compile-time type of $object$ to a given $type$.

The advantage of both variants is that the object structure remains clean, thus no additional methods and/or attributes are needed and full separation of the operation from the structure can be achieved. Consequently, the Visitable interface may be empty:

```
interface Visitable {}
```

The disadvantage of this approach is in fact that the type conversion is generally considered as a poor programming practice. However, the approach may still be viable when the others are unavailable.

4.1 Structured form

Let us first present a variant which is not object-oriented, thus it can be used (with some modifications) in programming languages not supporting object-oriented programming as well.

Each operation can be implemented with only one (possibly static) method. The method is responsible for processing each kind of element of object structure. To ensure correct processing of each element the type checking using instanceof construct is employed. Everything is usually packed in one (potentially long) conditional statement.

Additionally, type casting is used to access attributes and call methods specific to the processed element. In case of the composite element the recursion is typically used to proceed with the operation on composing parts of the element.

Let us present an example of the operation implementing size calculation of a given object structure:

```
static int size(Node node) {
   if (node instanceof Comment)
      return 0;
   else if (node instanceof PrintBold)
      return
         ((PrintBold) node).message.length();
   else if (node instanceof Print)
      return ((Print) node).message.length();
   else if (node instanceof Block)
      return size(((Block) node).first) +
             size(((Block) node).second);
   else ... // ERROR
}
```

It is obvious that the approach is not object oriented and that the length of conditional statement may be undesirably blown up. Additionally, the programmer must be aware of the common pitfall: the instanceof operator evaluates to true if the type of a given object, i.e., node in the above code excerpt, is equal to a given type or its supertype. In the above example if the PrintBold and Print conditions are switched the PrintBold part would never be executed since PrintBold is inherited from Print.

4.2 Object-oriented form

Now let us do the same but in a more object-oriented manner. The main idea here is to write the type checking and casting code only once and to implement each traversal operation within a single class.

First let us define the Visitor interface which represents an operation on object structure and declares a method for each type of element of the structure:

```
interface Visitor {
    void visit(Comment comment);
    void visit(Print print);
    void visit(PrintBold printBold);
    void visit(Block block);
}
```

Next we implement the abstract class AbstractVisitor which includes a method visit(Node node) in order

to be available to any element. The implementation of this method is similar to the conditional statement above, i.e., a set of instanceof operators.

However, now the processing of each element is delegated to the corresponding visit(...) method. Notice also, that type casting on the node argument must be used in order to ensure that the correct visit(...) method (specified in the Visitor interface) is called. Our example now looks like this:

```
abstract class AbstractVisitor implements Visitor {
   void visit(Node node) {
      if (node instanceof Comment)
         visit((Comment)node);
      else if (node instanceof PrintBold)
         visit((PrintBold)node);
      else if (node instanceof Print)
         visit((Print)node);
      else if (node instanceof Block)
         visit((Block)node);
      else ... // ERROR
   }
}
```

This version is arguably more elegant and readable, but the same pitfall (condition checking order) still applies.

Finally, the operation is implemented in a separate class which only needs to implement the methods of Visitor interface. For example, the size-calculation code excerpt is now:

```
class Sizer extends AbstractVisitor {
   int size;

   void visit(Print print) {
      size += print.message.length();
   }

   void visit(Block block) {
      visit(block.first);
      visit(block.second);
   }
}
```

In the following section we basically use the object-oriented form and take it to the next level by eliminating the type checking and casting at a cost of minorly polluting the object structure.

5 Visitor design pattern

Visitor design pattern is one of the most-used behavioral design patterns. The pattern is mainly used to implement an (traversal) operation to be performed on all elements of object structure. Its main advantage is to allow separation of the operation from the object structure while disregarding any explicit type checking and type casting. For a more thorough discussion see [2, 5, 11].

The pattern is based on the so called double-dispatch technique. Here the selection of the concrete method to execute is based on the run-time time types of two objects. In particular, it depends on the run-time type of the object (receiving the method call) and run-type type of the object, which is given as the first argument.

However, most of the object-oriented languages support only single dispatch. Nevertheless, the double dispatch can be simulated with two single dispatches. This approach is also used in the following example.

First let us present necessary additions to classes representing the object structure. Every class representing different elements of the object structure defines the so called accept(Visitor visitor) method which polymorphically accepts the visitor. Thus, we extend the Visitable interface:

```
interface Visitable {
   void accept(Visitor visitor);
}
```

Notice that, it is necessary that each class implementing Visitable overrides the accept(Visitor visitor) method. Notwithstanding, the implementations of overridden methods are all the same. In particular, the processing is delegated to the visitor by a simple call of the corresponding visitor's method:

```
void accept(Visitor visitor) {
   visitor.visit(this);
}
```

Such implementation is necessary as the run-time type of this is the same as its run-time type, thus the correct overloaded visit(...) method is called. Notice that, all operations implement the Visitor interface presented in the Section 4.2.

5.1 Classical form

Once the object structure is defined, a programmer must only implement the Visitor interface to define a new operation. Each method must specify two thinghs: the action to be performed , e.g., the size-calculation operation for a non-composite element Print:

```
void visit(Print print) {
   size += print.message.length()
}
```

and, optionally in case of a composite element also the access order in which composing elements are to be visited, e.g., the size-calculation of a composite element Block:

```
void visit(Block block) {
   block.first.accept(this);
   block.second.accept(this);
}
```

The visitation of composing elements is done via a call to the element's accept(...) method. Notice that the direct call to, e.g., visit(first) would be incorrect since first's compile-time type is Node and the appropriate method is non-existent in the visitor's class.

5.2 A simple refinement

As we see, in a composite object one must call the accept(...) method for each composing element of the object. A simple refinement to this is to additionally

create a base abstract class for all visitors, e.g., AbstractVisitor, implementing the following method:

```
void visit(Node node) {
   node.accept(this);
}
```

The idea is the same as in case of the method in Section 4.2. There a conditional statement and explicit type checking is used, while here the object-oriented polymorphism takes care for correct dispatching.

5.3 Visitor notes

Visitor design pattern is in general an elegant solution to implementing the operation on object structure as it enables the full separation. To explain the pattern we use the following code fragment which pretty prints (e.g. the Printer visitor) the given program (AST):

```
Visitable prog = new Block(...);
Visitor visitor = new Printer();
prog.accept(visitor);
```

The compile-time type of prog is Visitable and its run-time type is Block. Similarly, compile time and run time types of visitor are Visitor and Printer, respectively. See also Table 1.

variable	compile-time type	run-time type
prog	Visitable	Block
visitor	Visitor	Printer

Table 1: Compile-time and run-time types of variables.

The printing operation on the given object structure is started with a call to prog.accept(visitor). Since Java supports subtyping polymorphism (via virtual methods) actually the Block's accept(visitor) method is called (i.e., not Visitable's). The first method's argument is of type Visitor as it is the type of visitor. This method call is also referred as the first dynamic dispatch.

The second dynamic dispatch is as follows. The accept(visitor) method delegates its work via a call to visitor.visit(this). Because of subtyping polymorphism the Printer's visit(this) method is invoked. Notice that, the compile-time type of this is Block (not Visitable as it was before the method's call). Additionally, method overloading is also used here.

A disadvantage of visitor design pattern is that the accept(...) method must be present in the object structure. In some cases, e.g., external libraries and frameworks, such method may not be present. The other drawback is that the object structure is not clean. However, one can argue that the pollution is minimal.

6 Visitor by reflection

In this section we present a flexible approach to traverse the object structure using the programming language reflection capabilities. The reflection is a program's ability to examine (or even modify) itself [12]. Hence, this approach is infeasible in languages not supporting reflection mechanism (i.e., C, C++).

Similarly to the refinement technique described in Section 5.2 we use a base abstract class for visitors, e.g., AbstractVisitor. In this class we implement the visit(Node node) method using the reflection to invoke the corresponding visit(...) method. The basic implementation of AbstractVisitor.visit(Node node) is as follows:

```
void visit(Node node) {
   Method method = findVisitMethod(
                  getClass(), node.getClass());
   if (method == null) return;
   try {
      method.invoke(this, node);
   } catch (InvocationTargetException e) {
      ...
   } catch (IllegalAccessException e) {
      ...
   }
}
```

The corresponding visit(...) method is found via a call to findVisitMethod(...) which searches the classes in visitors class hierarchy. The method accepts the starting class of visitor and the starting class of node. The graphical demostration is provided in Figure 1. In the figure classes of visitors and their inheritance is shown on the left-hand side and visit(...) methods for each visitor are specified on the right-hand side (the type of first argument is shown).

We distinguish two basic directions in which the search may proceed.

In breadth The given visitor's class is examined for the corresponding visit(...) method. However, the search varies in the type of first argument. In particular, the run-time class of argument and, afterwards, its super classes are checked.

For example, in Figure 1 the search for visit(PrintBold) in a visitor of type HtmlPrinter one finds visit(Printer).

In depth The given visitor's class and, afterwards, its ancestors are examined for the corresponding visit(...) method.

For example, in Figure 1 the search for visit(PrintBold) in a visitor of type HtmlPrinter, one finds it later in its superclass XmlPrinter.

In the following subsections, we describe four search variants based on these directions:

- breadth-only search (the receiver's class hierarchy is ignored),
- depth-only search (the argument's class hierarchy is ignored),
- breadth-first search (both class hierarchies are explored, breadth direction first),

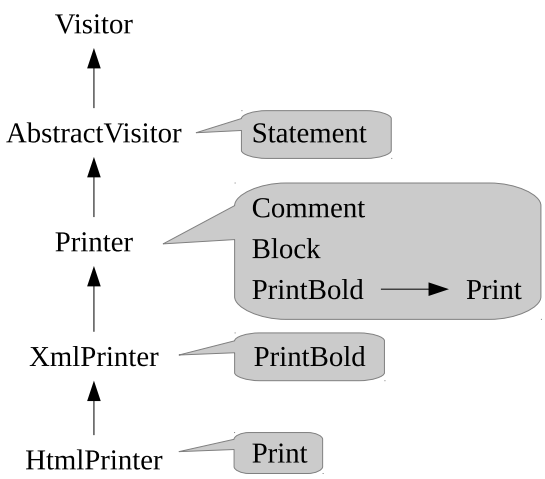

Figure 1: An example object-structure class hierarchy with provided visit(...) methods for each class.

- depth-first search (both class hierarchies are explored, depth direction first).

With reflection one can do much more: automatically visit also all members of visited object. See [6] for an example.

6.1 Breadth-only search

In breadth-only search the method-call receiver's class hierarchy is ignored and only the method's first argument is used. The search is implemented with the following method:

```
Method findVisitBreadth(Class visitorClass, Class
     nodeClass) {
   while (checkClasses(visitorClass, nodeClass)) {
      try {
         Method m = visitorClass.getMethod("visit",
              new Class[] { nodeClass });
         if (m != null) return m;
      } catch (NoSuchMethodException e) {}
      nodeClass = nodeClass.getSuperclass();
   }
   return null;
}
```

The reflection is employed via the getMethod() method which returns the visit(...) method of which the first argument type is a given class, e.g. nodeClass.

Notice that checkClasses(...) is a helper function which checks the given classes are not null and they are not at the bottom of class hierarchy (to prevent unnecessary errors). The implementation is as follows:

```
boolean checkClasses(Class visitorClass, Class
     nodeClass) {
   return visitorClass != null &&
         visitorClass != Visitor.class &&
         nodeClass != null &&
         nodeClass != Node.class;
}
```

6.2 Depth-only search

In depth-only search the method-call receiver's class hierarchy is examined, but the method's first-argument class hierarchy is ignored. The code for the depth-only search is basically the same as before except the line

```
nodeClass = nodeClass.getSuperclass();
```

which is replaced with

```
visitorClass = visitorClass.getSuperclass();
```

6.3 Breadth-first search

The breadth-first search is similar to breadth-only search. However, in case the method is not found in a given receiver's class, its ancestors are examined also. The techniques is implemented with the following code:

```
Method findVisitBFS(Class visitorClass, Class
     nodeClass) {
   while (checkClasses(visitorClass, nodeClass)) {
      Method m = findVisitBreadth(visitorClass,
           nodeClass);
      if (m != null) return m;
      visitorClass = visitorClass.getSuperclass();
   }
   return null;
}
```

6.4 Depth-first search

Finally, the implementation of the depth-first search technique is shown. It is similar to the breadth-first search except that the call to findVisitBreadth is replaced with findVisitDepth (depth-only search). Additionally, the line

```
visitorClass = visitorClass.getSuperclass();
```

is replaced with

```
nodeClass = nodeClass.getSuperclass();
```

6.5 Implementing visitor

The implementation of a traversal operation is straightforward, i.e., one only needs to extend Visitor or AbstractVisitor class and implement the corresponding visit methods. For example the execution of AST is done like this:

```
class Executor extends AbstractVisitor {
   void visit(Comment comment) {}

   void visit(Print print) {
      System.out.println(print.message);
   }

   void visit(PrintBold printBold) {
      System.out.println("**" + printBold.message +
           "**");
   }

   void visit(Block block) {
      visit(block.first);
```

```
    visit(block.second);
  }
}
```

7 Discussion

In this section we discuss the presented approaches for implementation of traversal operation. In particular, all nine described implementations (namely one using dedicated methods, two using type casting, two using visitor pattern, and four based on reflection) are compared using eleven selected criteria. Most of criteria are focused on technical aspects, while some aim on practical use of implemented solutions. Together they emphasize the important specifics of each approach. Next a more detailed discussion is presented. The summary of comparison is given in Table 2.

Object orientation All approaches but the first form of type casting (i.e., structured form) exploit OO capabilities of programming language. Consequently, only this form may be used (with some minor modifications) in a language without OO capabilities. Notice that, if instanceof operator is not supported one may simulate it via a special tag field [13].

Structure cleanliness Several approaches require additional methods to be provided in the implementation of object structure. More precisely, the dedicated-methods approach clearly demands separate implementation of many methods – one in each class of the class hierarchy representing the object structure, while approaches based on visitor pattern require additional accept() method. Both forms of type-casting and reflection-based approaches leave the object structure intact.

Operation/structure separation All approaches except the dedicated methods support the separation of implementations of traversal operation and object structure. As a result the source code of operation and structure can be independently developed [11].

Operation implementation To implement the traversal operation the code in dedicated-methods approach is dispersed over multiple classes. The structured type casting requires one method, while all other approaches require one class (with multiple but possibly simple methods) for the operation implementation.

Recompilation When a new traversal operation is added or changed only the approach using dedicated methods requires a recompilation of source code containing the implementation of operation. Notice that, this is a direct consequence of separation property.

Commence traversal To commence the traversal operation there are basically two options: to invoke an appropriate method (e.g., dedicated methods and first variant of type-casting) or to create a new class (and invoke one of its methods).

Special features The use of type checking and type casting is generally considered an undesired programming practice which often causes unexpected run-time type/cast errors [14]. Obviously, the type-casting approaches cannot avoid their use while all others can. Similarly the last approach uses reflection which is often considered advanced feature [12].

Dispatch technique For the dedicated-methods approach only single dispatch is needed. All other approaches need double dispatch (traditional via accept() method, via type casting, or via reflection).

Performance The performance of the approach is represented with the time required for completing traversal operation. Most of the approaches presented perform really well. The only exception here is the reflection-based approach which is considerably slower (for the obvious reason the dispatch is implemented by the programmer herself). On the other hand, this is the price paid for the flexibility the approach offer. Still, due to the rapid development of compilers this may not be a mayor pitfall. See [6] for more details on the performance.

Programming skills From the point of view of programming skills required to implement an approach anew the presentation order in the paper is from the simplest to the most complex one. We classified the approaches into three categories: easy, medium, and advanced, based on the level of programmer's understanding of concepts used.

Maintenance skills We argue the maintenance (i.e., changing existing operations and structure) is easier for the approaches that require more advanced programming skills as these approaches exhibit clear implementation rules. Notice also, that both casting approaches require the programmer to carefully plan the order of type checking.

Extensibility The performance in this criterion is a direct consequence of other technical criteria such as structure cleanliness and separation. To extend the object structure or to add another operation is usually more involved in approaches without clear separation of structure and operations and easier in the other. For the visitor design pattern and reflection approaches one only needs to create new class with several visit(...) methods.

	dedicated methods	casting (structured)	casting (OO form)	pattern (classical)	pattern (refinement)	reflection
object oriented	y	n	y	y	y	y
structure cleanliness	methods	clean	clean	accept()	accept()	clean
operation/structure separation	n	y	y	y	y	y
operation implementation	classes	method	class	class	class	class
recompilation needed	y	n	n	n	n	n
operation commence	invoke	invoke	create	create	create	create
special features	-	type cast	type cast	-	-	reflection
dispatch technique	single	via casts	via casts	double	double	via reflection
performance	fast	fast	fast	fast	fast	slow
programming skills	easy	medium	medium	advanced	advanced	advanced
maintenance	medium	hard	hard	easy	easy	easy
extensibility	hard	hard	hard	easy	easy	easy
suitability	simple	simple	simple	complex	complex	complex

Table 2: A summary of comparison

Suitability Finally, the approaches should be considered in the light of the problem they are solving. Hence, simpler approaches may be suitable for problems involving similar object structures with a small number of operations, where the use of more demanding approach may be even considered as over design (the principle of Occam's Razor may apply). On the other hand complex problems involving large object structure with multiple operations demand more advanced and flexible approaches.

8 Conclusion

In the paper we presented a simple working example to review a number of selected approaches for design and implementation of an object structure supporting a traversal operation. Apart precise description of more commonly known techniques we presented a reflection-based approach and several of its variants. The main outcome of the paper is a valuable comparison of all approaches based on eleven criteria, which enable the interested programmers to deeper understand and use the presented techniques in practice.

References

[1] Sommerville, I. *Software Engineering, Eighth Edition*. Pearson Education Limited, Harlow, England, 2007.

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, USA, 1995.

[3] The Groovy programming language page, `http://www.groovy-lang.org/design-patterns.html`, accessed: May 19th 2015.

[4] Mihelič, J; Dobravec. T. SicSim: A Simulator of the Educational SIC/XE Computer for a System-Software Course, Computer Applications in Engineering Education 23: 137-146, 2015.

[5] Nordberg III, M. E. The Variations on the Visitor Pattern. In *Proceedings of the PLoP'96 Writer's Workshop*, 1996.

[6] Palsberg, J., Jay, C. B. The Essence of Visitor Pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference - COMPSAC'98*, pages 9-15, Vienna, Austria, 1998.

[7] Sebring, J. Reflecting on the Visitor Design Pattern. *Java Report*, March 2001.

[8] Mai, Y., de Champlain, M. Reflective Visitor Pattern. In *Proceedings of the PLoP'2001 Writer's Workshop*, 2001.

[9] Appel, A. W. *Modern compiler implementation in Java*, 2nd ed. Cambridge University Press, 2002.

[10] Mihelič, J. VisitorFun source code, `https://github.com/jurem/VisitorFun`, accessed: April 1st 2015.

[11] Oliveira, Bruno C. d. S. and Wang, Meng and Gibbons, Jeremy. *The VISITOR Pattern as a Reusable, Generic, Type-Safe Component*, Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, 2008.

[12] Forman, I. R., Forman, N. *Java Reflection in Action*. Manning Publications, Shelter Island, USA, 2004.

[13] Schreiner, Axel-Tobias. *Objektorientierte Programmierung mit ANSI C*, Hanser, München, 1994.

[14] Pierce, Benjamin C., *Types and Programming Languages*, The MIT Press, 2002.