Reductor

A Pattern to Solve Problems with Decrease-And-Conquer Algorithms

J.Burgos[1] J.Galve J.García M.Sutil[2]

Universidad Politécnica de Madrid
Campus de Montegancedo, s/n Boadilla del Monte - 28660- Madrid
Phone: 34-1-3367455 Fax: 34-1-3367412
{jmburgos, jgalve, juliog}@fi.upm.es, msutil@arrakis.es

Abstract

Problems that require a traversal on a collection of data can be solved following a decrease-and-conquer algorithm strategy. *Reductor* is a behavioral pattern that describes an object-oriented design relying on higher-order facilities to solve this sort of problem.

Intent

Identify, encapsulate and organize the variant and invariant parts (both behavior and structure) of the *decrease-and-conquer* algorithm design technique[3] for application with programs that want to apply the technique to collections.

Also Known As

Decrease-And-Conquer Abstract Solver, Reduction Pattern.

Motivation

Let's consider a simple XML[4] document, called *library.xml*, as shown in Figure 1. The XML document represents a set of book records in a virtual library. The information provided for each book record is the title, URL address where the book is located, and a brief text

[1] This work has been partially supported by the Spanish project F.G.UPM-43700000190.

[2] This work has been partially supported by WorldNet 21 Ltd.

[3] The *Decrease-And-Conquer* algorithm design technique [LEV99], also known as *induction* [MAN96], *simplification*, or *reduction* [BRA96] consists of solving a problem by reducing its instance to a smaller one, solving the latter (recursively or iteratively) and then extending the obtained solution to get a solution to the original instance.

[4] XML is an extensible markup language that offers the opportunity to create richer documents than HTML can produce by introducing appropriate tags. Because of this flexibility, XML is considered to be the next generation markup language for general documents. More information about XML can be found in [MTU99].

description. Now we outline the problem of collecting the set of books that include a given word (or text pattern) in their description field. For example:

Collect all the books whose title includes the word "man"

Let's suppose we are using a XML parser for Java, and the previous XML document generates

```
<?xml version="1.0"?>
<!DOCTYPE library SYSTEM "library.dtd">
<library>
  <book>
    <title>El Quijote</title>
    <url>http://amazon.com/quijote.html</url>
    <description>Very interesting ...</description>
  </book>
  <book>
    <title>La Galatea</title>
    <url>http://amazon.com/galatea.html</url>
    <description>Interesting ...</description>
  </book>
  <book>
    <title>La Celestina</title>
    <url>http://amazon.com/celestina.html</url>
    <description>Interesting ...</body>
  </book>
  ...
</library>
```

Figure 1. Example of a XML format document

the corresponding DOM[5] tree (figure 2). Now, we can process this tree to search for the nodes that meet the above condition. As a result, we obtain a collection of nodes. This collection could be represented using arrays of nodes or grouped as a new tree (in order to generate a new XML file again). Each representation would force us to implement a different solution. However, this new implementation could be overcome if we abstract the way that solutions are combined (once again, with the help of the common abstract interface provided by a *Function Object*).

We can also think of different ways for showing the solution output. Suppose you are constructing a software application involving XML management, and you have to return different external representations from a XML document, for instance, another XML format file for a remote computer, an HTML file for a web-browser, or WAP[6] code for cellular phones. Because of the different output formats, *a priori*, one could decide to write a different

[5] DOM defines a set of Java interfaces to create, access, and manipulate internal structures of XML documents. XML is a language for describing tree-structured data. In XML, an element is represented by a start tag and an end tag. An element may contain one or more other elements between its start and end tags. Thus an entire document is rendered as a nested tree [MTU99].

[6] The *Wireless Application Protocol (WAP)* is the *de facto* worldwide standard for providing Internet communications and advanced telephony services on digital mobile phones, pages, personal digital assistants and other wireless terminals [WAP].

algorithm (i.e. a different implementation) for each problem. Alternatively, we might decide to abstract the function that is applied to the input data in order to obtain any abstract output format. As a consequence, the algorithm can remain as a common invariant part for the three problems.

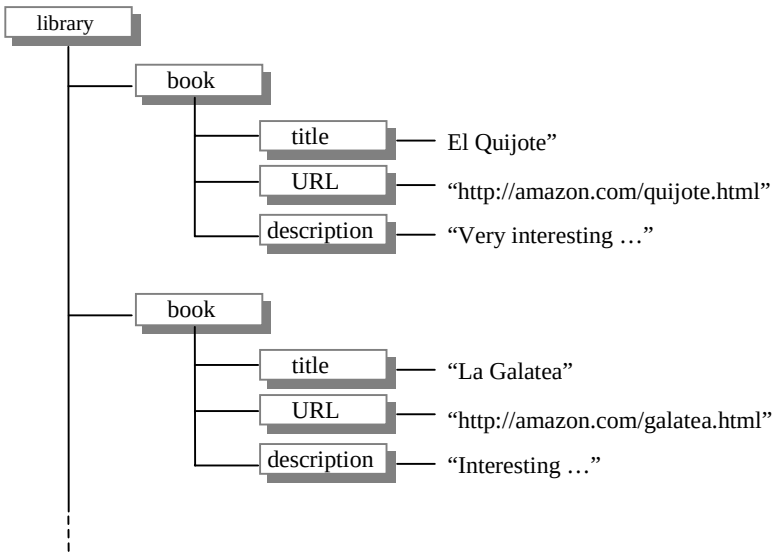

Figure 2. Example of a DOM tree format

Four primary components underly the above solution:

1. **InputIterator** decouples the input data representation and the algorithm. It is advanced one by one in an explicit loop.
2. **Expression** abstracts the function that must be applied to every item in the collection that meets the filter. It is a varying expression that can be combined with the traversal algorithm by placing it in the loop.
3. **Predicate** abstracts the predicate that the items in the collection must verify. It is a varying property that can be combined with the traversal algorithm by placing the predicate in the loop.
4. **Output** decouples the output data representation and the algorithm.

How do you combine these four components? Three different design strategies are possible:

a) A specific algorithm.
b) A general higher-order algorithm describing a family of solutions.
c) A half-way algorithm between a) and b), combining both general and specific components.

The first approach leads us to write a tailor-made algorithm whose implementation would probably try to exploit the efficiency with detriment to the reuse. It is the right decision if we do not plan to face new similar problems. The second approach promotes the definition of a common algorithmic skeleton to solve a wide range of problems. This second solution is more focused on the reuse and forthcoming variations of a current problem will take great advantage of it.

As is well known, one way to achieve behavior parameterization is to use an *external* iterator. Then, the varying property can be combined with the traversal algorithm by placing the function in an explicit loop that advances the external iterator one by one. As a result, the number of explicit loops corresponds to the number of member functions uses. However, there are good reasons to write such a loop once (see "Write a Loop Once" [MAR94] and the discussion in the Iterator pattern [GHJV95]) in a solution in which the traversal algorithm and the functions are combined by means of dynamic binding of the abstract function methods (*Filter*, *Expression* and *Output*). Disadvantages aligned with this approach are widely described in [KUH99].

Applicability

Use the Reductor pattern in the following situations:

- To develop solutions for problems that can be decomposed as a decrease-and-conquer algorithm. The pattern provides a general implementation for the common invariant part of the algorithm schema for free, leaving up to subclasses the implementation of the parts that can vary.
- To carry out proofs about correctness and efficiency of decrease-and-conquer based algorithms. The pattern can behave as a framework of algorithmic analysis. The clear separation between the common abstract-level and concrete input/filter/expression/output elements can help to get better-structured and data-independent algorithms, suitable to be reused or analyzed. Given an algorithm expressed as an instance of *Reductor*, its implementation is distributed along the components of the pattern. Then, it is possible to do some local analysis, firstly, and afterwards to reassemble these local results in a unique general result.

Structure

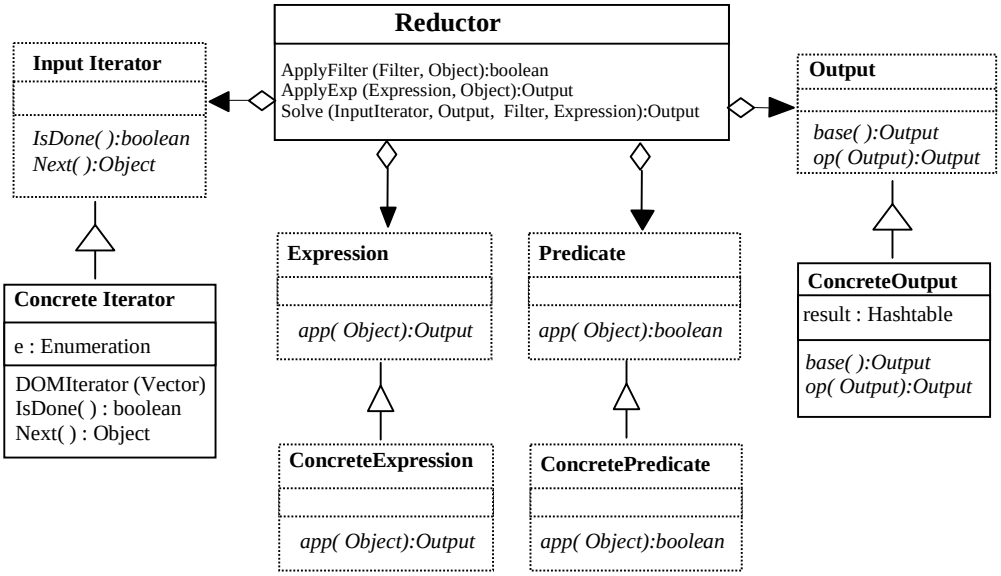

Figure 3. The Reductor pattern (structure)

Participants

- **Reductor** defines the operation `Solve`, a template method that implements a common algorithm for all decrease-and-conquer problems. It defines the methods `ApplyExp` and `ApplyFilter` that allow the *Expression* and *Predicate* to be declared as parameters.
- **Predicate** defines an abstract interface for predicate application. It is called from the operation `Solve` to filter elements in the decrease-and-conquer algorithm.
- **ConcretePredicate (XMLPredicate):** It implements the abstract operation `app` defined on *Filter*, providing a specific filter predicate.
- **Expression:** It defines an abstract interface for the function application. It is called from the operation `Solve` to apply a function to each element in the decrease-and-conquer algorithm.
- **ConcreteExpression (XMLExpression)**: It implements the abstract operation `app` defined on *Expression*, providing a specific expression function.
- **InputIterator**: It defines an abstract interface to traverse the collection of data managed by the decrease-and-conquer algorithm. It is modeled as an Iterator pattern.
- **ConcreteInputIterator (DOM Iterator)**: It implements the abstract operations defined on *InputIterator*, providing a specific data iterator.
- **Output**: It defines an abstract interface to model the output data of the decrease-and-conquer algorithm. It is called from the operation `Solve` to determine the null value (i.e. the operation `base`) and the combination operator (i.e., the operation `op`) in the output domain.

- **ConcreteOutput (Words)**: It implements the abstract operations defined on *Output*, providing a specific output data domain.

Collaborations

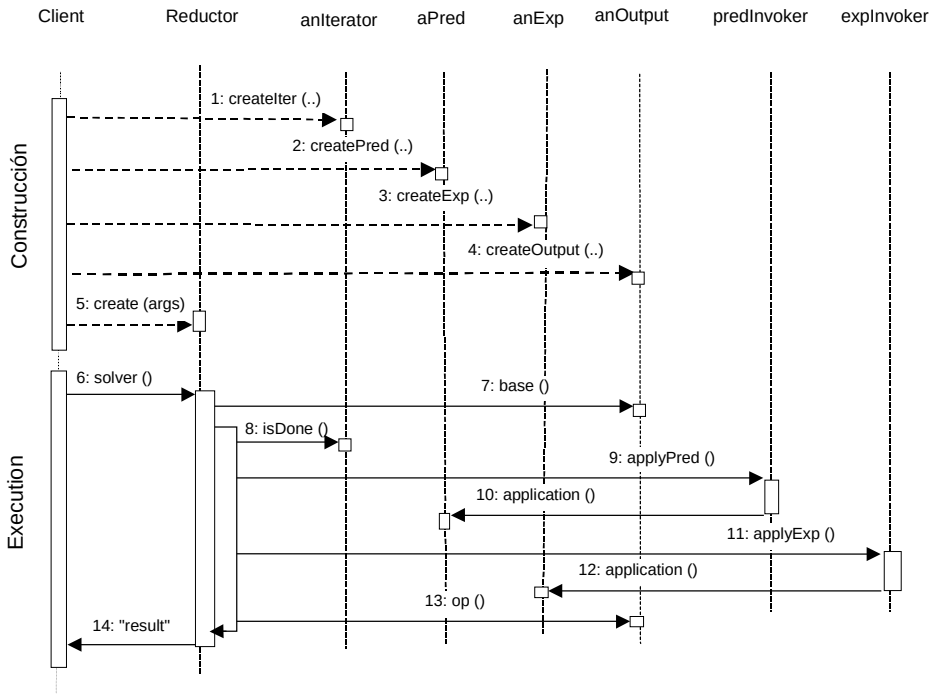

Figure 4. The Reductor pattern (collaborations)

- The client creates an iterator, a predicate, an expression and an output. He/she possibly supplies arguments for several of them.
- The client creates an instance of *Reductor* and supplies the above elements as arguments.
- The client invokes the method *Solve*, which encapsulates a while-loop implementation for the decrease-and-conquer algorithm strategy.
- The client invokes the method *Solve*, which encapsulates an implementation of the decrease-and-conquer algorithm strategy.

Consequences

The pattern presents the following advantages:

- *The Reductor pattern is a Higher-Order Function Object* [WIK] *for the decrease-and-conquer algorithm strategy.* It allows the definition of new algorithms based upon the decrease-and-conquer schema. The pattern's structure strongly establishes which components define the decrease-and-conquer and their relationships. Besides, the relationships with concrete components of the pattern (subclasses) are established.
- *The pattern can be used to build algorithm libraries*, in which the common behavior supplied by the algorithm design schemas can be reused.
- *The pattern stresses the so-called "Hollywood principle"* [GHJV95] since the operation *Solve* fully provides the engine for the schema. The pattern is a good example of "Write a Loop Once" [MAR94].
- *The pattern promotes an error-free strategy to develop decrease-and-conquer algorithms.* The operation *Solve* in the *Reductor* calls only those operations provided by *InputIterator*, *Output*, *Predicate* and *Expression*. However, to implement these operations, other operations defined by *ConcreteInputIterator*, *ConcretePredicate* and *ConcreteExpression* - or by other components- can be used.
- *The pattern provides a systematic approach that simplifies the development of decrease-and-conquer algorithms.* The way to build new algorithms can be tackled in two steps: first, defining the components that are related to the input and output data of the problem (InputIterator and Output), secondly, stating the behavioral components of the concrete algorithm to solve (Predicate and Expression). In both cases, all the components must be overridden by the concrete operations in the subclasses. To reuse the schemas effectively, subclass developers must understand which operations are defined for overriding.

The pattern presents the following disadvantages:

- It yields to inefficient implementations.
- It provokes communication overhead between *Reductor* and the other components.
- It yields a more complex programming model.

Implementation

Let us note the following implementation issues:

1. *Implementation-driven guidelines*. Operations involved in *Reductor* are defined by *InputIterator*, *Predicate*, *Expression* and *Output*.
2. *Primitive operations.* Operations defined in InputIterator, Predicate, Expression, and Output are primitive. Thus, they must be overridden. For example, they could be declared as pure virtual (in C++ conventions) or as part of an interface (Java conventions). The operation *Solve* must never be overridden.

3. *ApplyFilter and ApplyExp* are Template Methods [GHJV95] responsible of invoking *Filter* and *Expression*. Actually, these methods hide strategy patterns.
4. The operation *Solve* is a Template Method.

Sample Code

Next, we use the pattern to solve the problem of processing a XML document, just the way it was described in the Motivation.

1. Firstly, we propose an implementation for the class Reductor, which provides the skeleton for the decrease-and-conquer schema:

```
class Reductor
{
    Iteration collection;
    Output od;
    Predicate filter;
    Expression expression;
    void create (InputIterator collection, Output od,
                Predicate filter, Expression expression)
     {...}

     private boolean ApplyFilter (Item arg)
     {
           return filter.application (arg);
     }
     private Output ApplyExp (Item arg)
     {
           return expression.application (arg);
     }
     public Output Solve ()
     {
          Item thisItem;
          Output result = od.base();
          while (!collection.IsDone())
          {
               thisItem = collection.Next();
               if (ApplyFilter (thisItem))
                   result.Op (ApplyExp (thisItem));
          };
          return result;
     }
} // Reductor  class
```

2. Secondly, we implement a XMLIterator to process the DOM tree obtained from parsing the XML document. We use a *preorder* traversal in the flattening process:

```
class XMLIterator implements Iterator
{
     Enumeration e;
     public XMLIterator (String file)
     {
          Vector v = new Vector();
          Parser parser = new Parser(file);
          try {
```

```
                processing(parser.readStream(new FileInputStream(file)),v); }
            catch (FileNotFoundException e) {
                System.out.println("file not found"); }
            e=v.elements();
        } // XML_Iterator

        private void processing (Node node,Vector v)
        {
            NodeList nodeList = node.getChildNodes();
            for (int i =0; i < nodeList.getLength(); i++)
            {
              Node n = nodeList.item(i);
              v.addElement(n);
              processing (n,v);
            }
        } // processing

    } // XMLIterator class
```

3. Now, we implement a concrete XMLPredicate that will be used as a filter by the *Reductor* pattern. In our case, the XMLPredicate tests if a book record includes in its title the word "adventure".

```
class XMLPredicate implements Predicate
{
     String key;
     public XMLPredicate (String key)
     { this.key = key; }

     public boolean application (Item arg)
     {
            Node n = (Node)arg.value;
            if (n.getParentNode() != null)
            {
                if (n.getParentNode().getNodeName().equals("title"))
                   if (contains(n.getNodeValue())
                        return true;
            }
            return false;
     }

     private boolean contains(String line)
     { ... }

} // XMLPredicate
```

4. Given a DOM-tree node, there are two methods to go back and forth among child nodes. In our example, we use this functionality provided by the DOM API to implement the XMLExpression, by collecting the values of the sibling nodes (i.e., the URL and the book description), as follows:

```
class XMLExpression implements Function
{
   public Output application (Item arg)
   {
        Node n = (Node)arg.value;
        Words w = new Words();
        w.Base();
        Node parent = n.getParentNode().getParentNode();
```

```
         boolean enc = false;
         for (Node child = parent.getFirstChild(); !enc;
                        child = child.getNextSibling())
         {
            if (child.getNodeName().equals("author"))
            {
              Node node = child.getFirstChild().getNodeValue()
              String value = child.getFirstChild().getNodeValue();
              w.words.put (node, value);
              enc = true;
            }
         }
         return w;
      } // application (method)
  } // XMLExpression (class)
```

5. Finally, we have to keep the result of applying XMLExpression to the book records matching the XMLPredicate. Then, we are going to use a Hashtable structure to process the partial solutions of the problem.. Below, it is presented the class "Words" that is in charge of collecting and returning the solutions to the problem. It is straightforward to see that solutions are sorted and non-duplicated.

```
class Words implements Output
{
     public Hashtable words;
     public Words ()
     {
         words = new Hashtable();
     }

     // .. other methods

     public Output Op (Output x)
     {
          Words w = (Words)x;
          Enumeration e = w.words.elements();
          while (e.hasMoreElements())
          {
            // ...
          }
          return this;
     } // Op (method)
} // Words (class)
```

Known Uses

Higher-order functions are a very common programming mechanism in Functional Programming. On the contrary, traditional imperative languages give little scope for higher-order facilities: Pascal, Java and C allow functions as arguments, as long as those functions are not themselves higher-order, but have no facility for returning functions as results. In spite of this common drawback, there are several examples of higher-order facilities in object-orientation: the *blocks* in Smalltalk, the *components* in Eiffel, the *bound routines* in Sather. On its side, C++ is capable of returning objects that represent functions by overloading the

function application operator This underlies the genericity hailed in the C++ Standard Template Library (or STL [STKE86]) which requires advanced features of the language to implement higher-order functions.

Both the C++ STL and, more recently, the Java Generic Library (or JGL [JGL]) include a representative set of examples on how to combine higher-order facilities and iterators. It is simple to see how many of the algorithm templates presented in those libraries are instances *Reductor*.

Again in the scope of functional programming, the list comprehension mechanism of languages like Haskell [THO99] is a reduced version of the Reductor pattern for the case of problems dealing with lists. The construction of Reductor would have the following appearance in Haskell:

```
[Expression (x) | x <- iterator, Filter (x)]
```

where iterator is a Haskell list.

The Standard Template Library [STKE86], which was adopted as part of the standard C++ library, has a nice set of examples of the *Reductor* pattern in the non-mutating, minimum and maximum and generalized numeric algorithms sections. In the Iterator pattern [4] the operation Traverse in the template class ListTraverser can be modelled as a Reductor with an Expression named ProcessItem and without Filter. The operation Traverse in the class FilteredListTraverser can be modelled as Map quantifications with an Expression named ProcessItem and a Filter named TestItem. The Process Filters of the Booch's catalog of reusable software components in Ada [BOO87] are examples of application of the *Reductor* pattern. In [SOL85], we find an example of the *Reductor* pattern with the chunks of programming knowledge named *plans*.

In addition, it is straightforward to see how the application of a SELECT on a relational database can be expressed as an instance of the *Reductor* pattern, as follows:

```
SELECT average (salary-taxes) FROM employer WHERE type = 'Teacher';
```

- InputIterator: the rows of the EMPLOYER table.
- Predicate: the predicate that selects rows whose column is equal to "Teacher".
- Expression: the result of substracting the taxes to the salary.
- Output: the collection to accumulate the values: *average (salary-taxes)*

The next example shows how a JOIN sentence can be expressed as a Reductor again:

```
SELECT name, grade, subject FROM student, school-record
      WHERE id-number = '508' GROUP BY level
```

- InputIterator: the Cartesian product of STUDENTS and GRADES
- Predicate: Select the rows of the JOINED tables which ID-NUMBER is equal to "508".
- Expression: Extract the columns of NAME, GRADE, SUBJECT, LEVEL

- Output : Merge the answers (the columns) according to the LEVEL.

Finally, the Monads model [THO99] and the Quantification [BUR00] are good examples of a more formal usage of the *Reductor* pattern.

Related Patterns

- Command: Unlike Predicate and Expression, the Command pattern [GHJV95] defines a procedure object that does not take any arguments after creation and produces side-effects.
- Iterator: Predicate and Expression objects allow the use of data from inside (elements) and outside the collection.
- Observer: Instead of receiving a notification message from a subject, an observer may register call-back procedure objects at the subject that will perform necessary updates. This scheme adds another variant to the push-and pull-models of the Observer pattern.
- Composite: A Reductor object can be uniformly accessed with the Composite pattern [GHJV95]. A composite of Reductors can be operate as a pipeline structure, for which the Output result is the InputIterator argument for the next Reductor in the pipeline.
- Divide-And-Conquer and Backtracking patterns [BGGS98] are other good examples of patterns that model algorithm strategies.

Acknowledgments

Many thanks to our shepherd Dwight Deugo for his useful hints and comments on how to improve the paper. We would also like to express our deep appreciation to Eva González for her review of this article.

References

[BGGS98] Burgos et. al.: *An Approach to Algorithm Design by Patterns*, Proceedings of the 3rd European Conference on Patterns Languages of Programming and Computing (EuroPLoP'98), 63-76, 1998.

[BOO87] Booch G.: Software Components With Ada: Structures, Tools and Subsystems. Bejamin-Cummings, 1987.

[BRA96] Brassard G., T. Bratley, *Fundamentals of Algorithms*, Prentice Hall, 1996.

[BUR00] Burgos et al.: *Abstract Solution Design by Specification Refinement*, Innovation and Technology in Computer Science Education (ITICSE'2000) Helsinki, Finland July 11-13, 2000.

[GHJV95] Gamma, E. Helm R., Johnson R., Vlissides J. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[JGL] *JGL: The Generic Collection Library for Java.* URL: http://ultra64.cs.lafayette.edu/java/jgl/

[KUH97] Kühne, T., *The Function Object Pattern.* C++ Report, 9 (9) , October 1997, pp. 32-42.

[KUH99] Kühne, T. *A Functional Pattern System for Object-Oriented Design*. (PhD) University of KaisersLautern. URL: http://www.agce.informatik.uni-kl.de/~kuehne . 1999.

[LEV99] Levitin, A. *Do We Teach the Right Algorithm Design Techniques?* SIGCSE Bulletin and Proceedings, March 1999, pp.179-183.

[MAN96] Manber U., Introduction to Algorithms: A Creative Approach, Addison-Wesley, 1996.

[MAR94] Martin R.: *Discovering Patterns in existing Applications*. In James O. Coplien and Douglas C. Schmidt editors, Pattern Languages of Program Design, pp. 365-393. Addison-Wesley, 1994.

[MTU99] Maruyana, H., Tamura, K., Uramoto, N.: *XML and Java: Developing Web Applications*, Addison-Wesley, 1999.

[SCH96] Schmidt, D.R. *Towards a Classification Approach to Design*. In Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96 (1996), vol. LNCS 1101, Springer-Verlag, pp. 62-84.

[SCH99] Schmidt D. A.: A Return to Elegance: The Reapplication of Declarative Notations to Sofware Design. G. Gupta (Ed.): PADL'99, pp. 360-364, Springer-Verlag, 1999.

[SOL85] Soloway, E. From Problems to Programs Via Plans: The Content and Structure of Knowledge for Introductory Lisp Programming. J. Educational Computing Research, vol. 1(2), 1985, pp. 157-172.

[STKE86] Stepanov A., Kershenbaum, A.: *The Standard Template Library* URL: http: // www.sgi.com / Technology /STL/. 1986.

[THO99] Thompson, S. *Haskell: The Craft of Functional Programming.* 2nd ed. Addison Wesley, 1999.

[WAP] W@p Forum. http://www.wapforum.org/

[WIK] WikiWikiWeb Pages - The Portland Pattern Repository: *Category Functional Programming*. http://www.c2.com/cgi/wiki?CategoryFunctionalProgramming.