Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa

Rudolf Ferenc[1], Juha Gustafsson[2], László Müller[1], and Jukka Paakki[2]

[1] Research Group on Artificial Intelligence, University of Szeged & HAS
Aradi Vértanuk tere 1., H-6720 Szeged, Hungary, +36 62 544145
ferenc@cc.u-szeged.hu; h633226@sirius.cab.u-szeged.hu

[2] Department of Computer Science, University of Helsinki
P.O. Box 26, FIN-00014 University of Helsinki, Finland, +358 9 191 44180
gustafss@cs.helsinki.fi; paakki@cs.helsinki.fi

Abstract. A method for recognizing design patterns from C++ programs is presented. The method consists of two separate phases, analysis and reverse engineering of the C++ code, and architectural pattern matching over the reverse-engineered intermediate code representation. It is shown how the pattern recognition effect can be realized by integrating two specialized software tools, the reverse engineering framework *Columbus* and the architectural metrics analyzer *Maisa*. The method and the integrated power of the tool set are illustrated with small experiments.

Key Words

design patterns, reverse engineering, source code parsing, C++, object-oriented design

1 INTRODUCTION

Due to the increase of size and complexity of software systems, the importance of being able to comprehend and assess the quality of (legacy) software code has been steadily rising. Traditional software metrics, such as complexity, cohesion, and coupling have not fully met the requirements of industrial software development, mostly because they are rather low-level concepts and do not capture the high-level design decisions actually made by the designers and programmers when constructing the software.

A more high-level view over a software system can be created by more modern techniques commonly known as *reverse engineering*. In reverse engineering, the objective is to extract the static structure and the dynamic behavior of the code into some abstract representation, so as to make it easier to explore the essential aspects of the system by ignoring insignificant implementation details. In the idealistic case the low-level code is reverse-engineered backwards into its original

design – or at least to a form that might have been the intent of the software designers.

Reverse engineering methods and tools produce a wide variety of abstract software representations. A natural and currently quite popular strategy of abstracting object-oriented programs is to extract them into a set of UML diagrams [9]. Under the assumption that UML is not just a general-purpose modeling language but also a language for describing *software architectures*, the generated diagrams can indeed be regarded as representing the architectural design of the system.

While the automatic generation of UML diagrams from software code is already supported by a number of reverse-engineering tools, it is somewhat surprising that one of the cornerstones of contemporary object-oriented software engineering, *design patterns* [3], is in almost total lack of advanced tool support. By abstracting practical solutions to frequently occurring design problems into an object-oriented format, design patterns are a most natural and useful asset when recovering the architectural design and the underlying design decisions from the software code.

In this paper we present a technique for automatically recognizing design patterns from object-oriented (C++) code. The method relies on two software tools, *Columbus* [1][2] and *Maisa* [8][10]. Columbus is a versatile reverse-engineering system that transforms C++ programs into a number of abstract representations, including UML class diagrams. Maisa is a metrics tool that analyzes the quality of a software architecture given as a set of UML diagrams. Since one of the functionalities of Maisa is the mining of design patterns from the input architecture, Columbus and Maisa together provide the combined effect of recognizing design patterns from C++ code: the code is first transformed by Columbus into UML class diagrams, which are then traversed and matched against a set of predefined design patterns by Maisa. The integration of Columbus and Maisa is technically straightforward: Columbus exports its UML diagrams into Maisa using its textual input format.

The Columbus-Maisa pair can be used both to document and analyze a software system implemented in C++. In addition to that, since the foremost application area of Maisa is the software design phase and that of Columbus is the implementation (coding) phase, the tools can be used to verify that the architectural design decisions (Maisa) are followed in the implementation phase and actually realized in the code (Columbus). This makes it possible to assess more closely the software development process as well as track the evolution of design decisions during it.

We proceed as follows. The metrics analyzer Maisa is presented in Chapter 2, concentrating especially on its pattern mining facility. The reverse-engineering system Columbus is presented in Chapter 3, followed by a short description of the tool integration in Chapter 4. In Chapter 5 we discuss our experiments on design pattern recognition. Finally, conclusions and future directions are addressed in Chapter 6.

2 MAISA

Maisa [8][10] is a software tool for the analysis of software architectures, developed in an ongoing research project at the University of Helsinki. The key idea in Maisa is to analyze design-level UML diagrams and compute architectural metrics for early quality prediction of the software system.

In addition to calculating traditional (object-oriented) software metrics such as *Number of Public Methods*, Maisa looks for instances of design patterns (either generic ones such as the well-known GoF patterns [3] or user-defined special ones) from the UML diagrams representing the software architecture. According to the experiences gained so far with industrial cases, the level of abstraction is crucial for the success of the analysis: the more detailed the diagrams are, the more accurate are the results. Therefore design pattern mining at the detailed level of source code, as presented in this paper, is a most promising way of improving the practical usability of Maisa.

Maisa also incorporates metrics from different types of UML diagrams and execution time estimation through extended activity diagrams [13]. Additionally, we are currently studying the possibility of using dynamic information (such as sequence diagrams) for defining patterns more accurately.

2.1 Constraint satisfaction in pattern mining

Constraint satisfaction [4][5] is a generic technique that can be applied to a wide variety of tasks, in our case to mining patterns from software architectures or software code. A constraint satisfaction problem (CSP) is given as a set of variables and a set of constraints restricting the values that can be assigned to those variables. *Unary constraints* (denoted as P_i) restrict the values for a single variable, while *binary constraints* (denoted as P_{ij}) represent a condition for a pair of variables. The CSP is often modeled as a graph, where the nodes represent the variables and the arcs represent the constraints.

Formally, a CSP can be stated as follows [4]:
$(\exists x_1)(\exists x_2)...(\exists x_n)P_1(x_1) \wedge P_2(x_2) \wedge ... \wedge P_n(x_n) \wedge P_{12}(x_1, x_2) \wedge P_{13}(x_1, x_3) \wedge ... \wedge P_{n-1n}(x_{n-1}, x_n)$,
with P_{ij} included for all $i < j$.

In practical terms, variable domains (D_i) must consist of a finite number of discrete values. Even so, the solution of trying out all combinations would be too slow. In addition, most combinations would make no sense, so it's no use to try them at all. We may try a particular value several times, even if there is no way that the value could be a solution for a given variable. Therefore we must find a way to effectively prune out impossible candidates.

It is not always possible or practical to find a complete solution. If we allow *partial satisfiability*, we may accept those solutions that violate (to a certain extent) some of the constraints. In this situation, the constraints do not offer just exclusive alternatives. We may define our criteria separately for each case. A disadvantage of this technique is that the number of potential solutions may go up quite rapidly. Some research has been done regarding the case of partial

satisfiability and it may suit our problem quite well, as the patterns themselves are not always well-defined (discussed further in Chapter 2.3).

We define our pattern mining problem as a CSP in the following way:

- The variables (nodes) represent the roles of a pattern.
- The variable domains are initialized to contain all the names (identifiers) in the diagram(s) in question.
- Unary constraints represent conditions for a single role (e.g. the element in role X must be of type abstract class).
- Binary constraints represent conditions between two roles (e.g. the class in role X must be a subclass of the class in role Y).

For each pattern we compute a result, i.e. the role bindings that describe this particular pattern. The number of these bindings depends on the pattern in question. A binding is a pair {*role*,*element*}, where *role* is the name of the role and *element* is the diagram element that appears in that role, e.g. in the **Factory Method** pattern [3] two of the roles are *Product* and *Creator*.

2.2 Reducing the search space

A simple and useful way of testing the candidate values is *backtracking*, where the conditions are tested for each value. If the conditions are not met, that value is discarded. Before backtracking, we must make sure that there are no unsuitable values in the domain of each variable. This means that if we require that a certain variable can only have class-typed values, then we can prune all attributes, methods etc. from its domain. This way we can make the number of candidates as small as possible. Currently we use the AC-3 algorithm [4] in Maisa, but the algorithm can be easily replaced. This implementation has originally been designed by Pauli Misikangas [6].

AC-3 algorithm The first and most trivial requirement is node consistency. Node i is *node consistent*, iff $\forall x \in D_i, P_i(x)$ holds. The following algorithm ensures node consistency.

```
procedure NC-1:
begin
    for i ← 1 until n do
        begin
            D_i ← {x ∈ D_i | P_i(x)}
        end
end
```

Thus, for example, all attribute-entities will be pruned by NC-1 from the domain of a variable having a constraint that allows only solutions of type *class*.

Arc consistency is defined in a similar fashion: Arc (i, j) is *arc consistent*, iff $\forall x \in D_i$ such that $P_i(x)$ holds, $\exists y \in D_j$ such that $P_j(y)$ and $P_{ij}(x, y)$. A more detailed discussion of arc consistency can be found in [7].

A single arc can be revised using the following procedure REVISE that returns a boolean value. The idea is similar to that behind node consistency. We delete all values from the domain of the originating node D_i, for which there are no 'legal' arcs:

```
procedure REVISE((i,j)):
begin
    DELETE ← false
        for each x ∈ D_i do
        if ∄y ∈ D_j such that P_ij(x, y), then
          begin
              delete x from D_i
              DELETE ← true
          end
    return DELETE
end
```

The AC-3 algorithm first utilizes the node consistency algorithm and then the arc consistency revision algorithm as follows. We denote the entire CSP graph with G and the respective set of arcs (constraints) with $arcs(G)$. Additionally we denote the current (non-consistent) set of arcs with Q, which means that the algorithm halts as soon as Q is empty.

```
procedure AC-3:
begin
    NC-1
    Q ← {(i, j)|(i, j) ∈ arcs(G), i ≠ j}
    while Q not empty do
      begin
        select and delete any arc (k, m) from Q
        if REVISE((k, m)) then
              Q ← Q ∪ {(i, k)|(i, k) ∈ arcs(G), i ≠ k, i ≠ m}
      end
end
```

After the domains have been made consistent, we search for correct bindings among the remaining values that satisfy the current set of constraints. In the simple case we have only one value for each variable.

Extensions Many design patterns are 'related' to each other in the sense that they have common elements (see e.g. metapatterns in [11]). These relationships may be taken advantage of in two ways: by ordered search of patterns and by the use of auxiliary facts [6]. This information is updated as the search proceeds, when a particular pattern is being searched, new facts are added. These facts can then be utilized later on when searching for other patterns. Consider, for example, that we are searching for instances of the *Abstract Factory* pattern

which has the *Factory Method* pattern as its prerequisite [3]. We would now take advantage of a new fact *Factory Method* that has been added while searching for instances of the *Factory Method* pattern.

2.3 Interaction

The AC-3 (and generally any other purely syntactic) algorithm may still produce a large number of false positives, when we have a non-trivial task like finding vaguely defined design patterns. To make matters worse, several fairly common design patterns have features that are very difficult or even impossible to model as a set of constraints. In these cases human intuition and insight is essential for verifying the potential bindings generated by the algorithm.

Many design patterns are too abstract to be easily represented syntactically [3]. The situation becomes even more complex if we require a fine-grained classification of separate pattern instances. Consider, e.g., the situation where finding instances of the metapattern **1:1 Connection** [11] is not enough, but we want to make the distinction between the patterns **Bridge** and **Command**. Their syntactic structure is alike so an attempt to automatically separate them would not be realistic.

Another related problem is that in many cases the design diagrams simply do not contain enough information. (UML) associations are a typical example. This concept has quite a lot of expressive power. An association can be implemented in a number of different ways. A common case would be to include an attribute in one of the classes containing a reference to the other class, or to have a class that calls a method of another class. During the design phase the more general representation is usually enough: either we do not know the implementation details, or we do not wish to fix them yet. However, in order to recognize many common design patterns (such as **Abstract Factory** and **Builder**), we need to know these connections explicitly. In these cases we either have to include more detailed information in the UML diagrams or try to find the patterns using incomplete information. The former alternative is not viable in practice, as in most cases we simply do not have (or even need) the required level of detail in the design phase. As a solution to the latter case partial satisfiability techniques might be worth investigating.

Even when dealing with correct positive instances of design patterns, the number of possible bindings can become large (e.g. when searching for **Composite** or **Mediator** patterns), since the number of elements that can participate in a certain role in a pattern is not limited. The basic CSP algorithm would try to find them all. This is also a situation, where human interaction is quite helpful.

An important issue is that the rules describing the patterns are correct. This is even more important, if the semantics of the pattern are complex. Missing or false constraints may either produce a number of false positives (which can be frustrating) or false negatives (which is what we would most likely to avoid). This issue might seem obvious, but considering the small semantical subtleties many patterns have, finding the correct representation for a pattern is not necessarily trivial.

To be of any use this kind of interaction naturally requires a highly knowledgeable user (knowledge of both the design patterns and the problem domain is essential). It must be emphasized, though, that interaction is usually not required, and the AC-3 algorithm produces results relatively fast even when working with larger domains.

Many features discussed here, such as the verification of potential bindings or the presentation of design patterns, require to extend the current user interface of Maisa. For the time being, only a textual presentation is available. In the future, more usable alternatives will be developed.

3 COLUMBUS

Columbus is a reverse engineering framework [1][2], which has been developed in cooperation between the Research Group on Artificial Intelligence in Szeged and the Software Technology Laboratory of Nokia Research Center. Columbus is able to analyze large C/C++ projects and to extract their UML class model [9] as well as conventional call graphs.

The main motivation for developing the Columbus system has been to create a general framework for combining a number of reverse engineering tasks and to provide a common interface for them. Thus, Columbus is a framework tool which supports project handling, data extraction, data representation, data storage, filtering, and visualization. All these basic tasks of the reverse engineering process for the specific needs are accomplished by using the appropriate modules (plug-ins) of the system. Some of these plug-ins are present as basic parts of Columbus, while the system can be extended to serve other reverse engineering requirements as well. This way we have got a really versatile and easily extendible tool for reverse engineering.

3.1 Overview of the Columbus System

The basic operation of Columbus is performed by three types of plug-ins:

- *Extractor plug-ins* (currently an extractor for C/C++), whose task is to analyze a given input source file and to create a file, which contains the extracted information.
- *Linker plug-ins*, whose task is to build up and filter the merged internal representation of the project. This process is carried out based on the files created by the extractor plug-in.
- *Exporter plug-ins*, whose task is to export the internal representation built up and filtered by the linker plug-in into a specific output format. (Currently: Maisa, TDE Mermaid 2.2, TED 1.0, Rational Rose, Microsoft Jet Database, HTML, XML and ASCII.)

In addition to the built-in plug-ins, the user can easily write and add his/her own new plug-in DLLs to the Columbus system using the *plug-in API*.

3.2 The Extraction Process

Columbus can handle projects consisting of input files along with their settings. The project is displayed in a tree view and it can simultaneously contain source files of different programming languages.

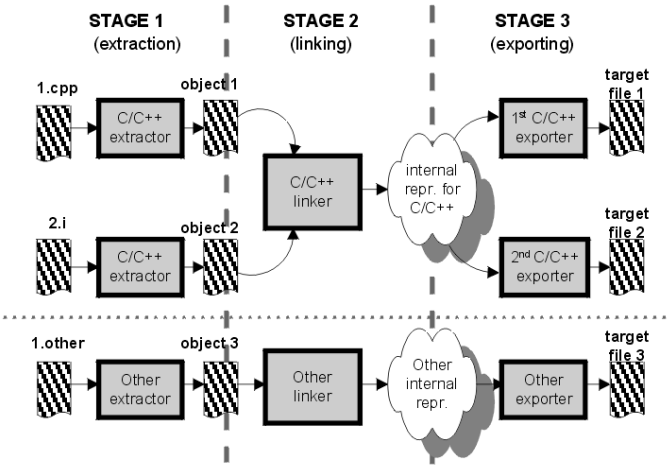

Fig. 1. The extraction process

The extraction process (Figure 1) itself is very similar to a compiler. The first stage is data extraction. Columbus takes the input files one by one and passes them to the appropriate extractor, which creates the corresponding internal representation files. In the second stage the linker plug-in is automatically invoked in order to link (merge together) the internal representation files in the memory. In the third stage the data is transformed into a given export format, usually based on a filtered internal representation. An important advantage of Columbus is that it can incrementally perform all these steps, that is, if the partial results of certain stages are available and the input of the current stage has not been changed, the partial results will not be recreated.

3.3 CAN – The C/C++ Analyzer

Parsing of the input source code is performed by the C/C++ extractor plug-in of Columbus, which invokes a separate program called *CAN* (**C**++ **AN**alyzer). CAN is a command-line (console) application for analyzing C/C++ code. This allows its *integration* into the user's makefiles and other configuration files, thus facilitating automated execution in parallel with the software build process.

Basically, CAN accepts one complete translation unit at a time (a preprocessed source file). However, for files that are not preprocessed a preprocessor

will be invoked. The actual results of CAN are the internal representation files, which are the binary saves of the internal representations built up by CAN during extraction.

One of the greatest assets of CAN is probably the *handling of templates* and their *instantiation at source level*, which is accomplished using a *two-pass technique* in program analysis. The first pass only recognizes the language constructs in connection with the templates (like a "fuzzy" parser) and instantiates them. The second pass then performs the complete analysis of the source code and creates its internal representation.

The C++ language processed by the analyzer covers the ISO/IEC standard from 1998 [12]. Furthermore, this grammar is extended with the Microsoft extensions used in Microsoft Visual C++.

4 INTEGRATION OF COLUMBUS AND MAISA

As mentioned in the previous chapter, Columbus offers an Application Programming Interface to access the information extracted from a C/C++ program. This API establishes a direct connection to the ASG (Abstract Semantic Graph) of the analyzed project, which is the common internal representation for all the information generated by the C/C++ extractor. This way it is very easy to create an exporter plug-in for Columbus that can transform the ASG into any desired data format.

Because Maisa is implemented entirely in Java, it cannot access Columbus's ASG directly, so we chose a trivial way for connecting the two tools: an exporter plug-in in Columbus creates a file in Maisa's input file format, which can then be opened and processed further with Maisa.

The file created by Columbus contains the reverse-engineered information in PROLOG format, as facts over the main program elements (classes, attributes, etc.) and their relationships (subclassing, etc.) This information is detailed enough to support, most notably, the automatic recognition of design patterns from the underlying C++ source code.

5 EXPERIMENTS

The design pattern recognition approach described above has been tested with a set of small experiments. For this purpose we have implemented some of the standard design patterns [3] in C++. After that we have used Columbus to analyze the code and to extract high-level structural information from it into the input format of Maisa. Finally, Maisa has been applied to recognize design patterns from the structural information (and, indirectly, from the original C++ code).

We demonstrate this process with the **Singleton** [3] design pattern as an example. The intent of this pattern is to ensure that a class has only one instance. One possible implementation of Singleton in C++ is as follows:

```
class MySingleton {
public:
  static MySingleton* getInstance();
protected:
  MySingleton() {};
private:
  static MySingleton* instance;
};

MySingleton* MySingleton::instance = 0;

MySingleton* MySingleton::getInstance() {
  if (instance==0) {
    instance=new MySingleton();
  }
  return instance;
}
```

The semantic intent of **Singleton** is realized by a static field that holds the only instance of the class. The constructor of this class is not accessible for other classes. The static *getInstance* method creates the single instance, if necessary, and returns it. The only way to access the instance of the class is through this method.

When analyzing this piece of code with Columbus, we obtain (UML specific) information over class relations, such as generalizations, aggregations, associations, as well as the calling dependencies. This information is generated by Columbus into the following PROLOG-like format:

```
class("MySingleton").
method("MySingleton.getInstance()").
public("MySingleton.getInstance()").
static("MySingleton.getInstance()").
has("MySingleton","MySingleton.getInstance()").
returns("MySingleton.getInstance()","MySingleton").
method("MySingleton.MySingleton()").
protected("MySingleton.MySingleton()").
has("MySingleton","MySingleton.MySingleton()").
attribute("MySingleton.instance").
private("MySingleton.instance").
static("MySingleton.instance").
has("MySingleton","MySingleton.instance").
typeof("MySingleton.instance","MySingleton").
```

On Maisa's side, the **Singleton** candidates are specified by the following facts:

```
class("Singleton").
attribute("Singleton.instance").
has("Singleton","Singleton.instance").
typeof("Singleton.instance","Singleton").
static("Singleton.instance").
```

This description states that a **Singleton** candidate (class) must have a static attribute whose type is the same as the class itself. When matching this pattern description with the high-level description of the C++ fragment, as produced by Columbus, Maisa produces the following output:

```
Solution 0
  Singleton.instance = MySingleton.instance
  Singleton = MySingleton
```

According to this, Maisa has found an instance of the **Singleton** pattern. The equations on the last two lines give the bindings generated by the AC-3 constraint satisfaction algorithm, with the name of the pattern role on the left-hand side of the equation, and the class, attribute, or method taking that role in the C++ code on the right hand side.

The following table summarizes the findings of our experiments. The table gives the names and brief descriptions of the design patterns [3] that have been recognized with the Columbus-Maisa couple.

Pattern name	Description	Missing facts
Singleton	Ensures that a class has only one instance	-
Visitor	Represents an operation on the elements of an object structure	-
Builder	Separates the creation of a complex object from its representation	reads(method,attribute) writes(method,attribute)
Factory Method	Defines an interface for creating subclass-specific objects	-
Prototype	Creates objects by cloning prototypical instances	-
Proxy	Provides a placeholder for an object to control access to it	reads(method,attribute)
Memento	Captures the state of an object	-

There are certain facts that are required for some design patterns but that Columbus does not generate yet. These facts are listed in the third column of the table. In the experiments, the additional facts were added manually to the

output which was then exported to Maisa. By this, Maisa was able to correctly recognize the corresponding patterns as well.

The facts `reads(method,attribute)` and `writes(method,attribute)` both mean that the specified method accesses the specified attribute. The fact `writes` has the additional meaning that the state of the attribute changes in a way.

6 CONCLUSION AND FURTHER WORK

We have presented a method and tool set for recognizing design patterns from C++ code. The method can be used for reverse-engineering purposes to study the structure, behavior and quality of the code, as well as for tracking the evolution of design decisions between the architectural level and the implementation level of a software system written in C++.

In our experiments it was noticed that some design patterns, like **Iterator** and **Observer**, cannot be recognized with the current method. The reason for this is that in the Maisa pattern library the descriptions of such patterns contain generated facts, i.e., structural facts that are dynamically pushed to the input by Maisa when it recognizes a particular kind of pattern or a special kind of a common class relation. In order to recognize these kinds of design patterns, our combined method must be extended with matching of the generated facts as well.

While our initial tiny experiments show the potential capability of the pattern recognition approach, more extensive experiments with real cases must be carried out to verify the real power of the method. Such larger-scale experiments have been made with another design pattern tool [6] (using the same pattern mining algorithm as Maisa), and the results show that the technique is capable of detecting most standard design patterns quite efficiently – even those that the original programmer did not explicitly design into the code. On the other hand, it was noticed that some very abstract and fuzzy patterns (such as **Interpreter**) cannot be reliably detected by automatic means and that the performance degrades with large software systems (consisting of hundreds of thousands of program lines).

Further work is also needed for separately improving the tools. The most important improvement on the Columbus side is extending the set of generated UML diagrams beyond the currently supported class diagrams, while the main development trends in Maisa are performance prediction with extended UML activity diagrams and visualization of the recognized design patterns on top of the subject software architecture.

Acknowledgements

Maisa is being developed in a research project financed by the Finnish National Technology Agency (Tekes), Nokia Research Center, Nokia Mobile Phones, Space Systems Finland, and Kone. In addition to the Finnish co-authors of this paper, the Maisa research group includes Lilli Nenonen, Minna Majuri, and Inkeri Verkamo.

References

1. Beszédes, Á., Ferenc, R., Magyar, F. and Gyimóthy, T. *Columbus Setup and User's Guide.* ©1998-2000 Nokia Research Center.
2. Ferenc, R., Magyar, F., Beszédes, Á., Márton, G., Tarkiainen, M. and Gyimóthy, T. *Columbus 2.0 - Tool for Reverse Engineering Large Object Oriented Software Systems.* Technical Report TR-2000-002, University of Szeged, 2000.
3. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
4. Mackworth, A. *Consistency in network of relations.* Artificial Intelligence 8,1 (1977), 99-118.
5. Mackworth, A. *The logic of constraint satisfaction.* Artificial Intelligence 58,1-3 (1992), 3-20.
6. Misikangas, P. *Automatic recognition of design patterns in object-oriented programs* (in Finnish). Master's Thesis C-1998-1, University of Helsinki, Department of Computer Science, 1998.
7. Mohr, R., Henderson, T. *Arc and path consistency revisited.* Artificial Intelligence, 28 (1986), 225-233.
8. Nenonen, L., Gustafsson, J., Paakki, J. and Verkamo, A.I. *Measuring object-oriented software architectures from UML diagrams.* In Proc. 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering. Sophia Antipolis, France, 2000, 87-100.
9. *OMG Unified Modeling Language Specification.* Version 1.3, ©1999 Object Management Group, Inc.
10. Paakki, J., Karhinen, A., Gustafsson, J., Nenonen, L. and Verkamo, A.I. *Software metrics by architectural pattern mining.* In Proc. International Conference on Software: Theory and Practice (16th IFIP World Computer Congress). Beijing, China, 2000, 325-332.
11. Pree, W. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.
12. *Programming languages – C++.* ISO/IEC 14882:1998(E).
13. Verkamo, A.I., Gustafsson, J., Nenonen, L. and Paakki, J. *Measuring design diagrams for product quality evaluation.* In Proc 12th European Software Control and Metrics Conference. London, England, 2001, 357-366.