Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems

Rudolf Ferenc[1], Ferenc Magyar[1], Árpád Beszédes[1], Ákos Kiss[1], and Mikko Tarkiainen[2]

[1] Research Group on Artificial Intelligence, University of Szeged & HAS
Aradi Vértanuk tere 1., H-6720 Szeged, Hungary, +36 62 544145
{ferenc,magyar,beszedes,akiss}@cc.u-szeged.hu

[2] Nokia Research Center, P.O.Box 407
00045 Nokia Group, Finland, +358 9 43766464
mikko.t.tarkiainen@nokia.com

Abstract. One of the most critical issues in large-scale software development and maintenance is the rapidly growing size and complexity of the software systems. As a result of this rapid growth there is a need to understand the relationships between the different parts of a large system. In this paper we present a reverse engineering framework called Columbus that is able to analyze large C/C++ projects. Columbus supports project handling, data extraction, -representation, -storage and -export. Efficient filtering methods can be used to produce comprehensible diagrams from the extracted information. The flexible architecture of the Columbus system (based on plug-ins) makes it a really versatile and an easily extendible tool for reverse engineering.

Key Words

reverse engineering, source code parsing, large-scale software systems, UML, Class Model, C/C++, templates, call graph.

1 INTRODUCTION

One of the most critical issues in large-scale software development and maintenance is the rapidly growing size and complexity of the software systems. As a result of this rapid growth there is a need to understand the relationships between the different parts of a large system [1] [2]. The substantial amount of existing legacy code and/or high number of the participants in code development also necessitates the use of tools for *reverse engineering* [14]. Reverse engineering is "the process of analyzing a subject system to (a) identify the system's components and their interrelationships and (b) create representations of a system in another form at a higher level of abstraction" [4].

In this paper we present a reverse engineering framework called *Columbus* [5], which has been developed in a cooperation between the Research Group

on Artificial Intelligence in Szeged and the Software Technology Laboratory of the Nokia Research Center. Columbus is able to analyze large C/C++ projects and to extract their UML Class Model [12] and call graph. It supports project handling, data extraction, data representation and data storage. Furthermore, efficient filtering methods can be used to produce comprehensible (clear-cut) diagrams from the extracted information. The flexible architecture of the Columbus system (based on plug-ins) makes it a really versatile and an easily extendible tool for reverse engineering.

The Software Technology Laboratory of the Nokia Research Center has developed a software product called *TED*[1] (**T**elecom Design Environment **ED**itor) for supporting geographically distributed, location-transparent collaboration of software designers and developers [18] [20]. TED supports UML modelling, therefore it was straightforward to export the diagrams created with Columbus into the TED repository.

In [2] four main assesment criteria (*analysis*, *representation*, *editing/browsing* and *general capabilities*) have been introduced to compare different reverse engineering tools. In the following we briefly investigate the Columbus system using these criteria.

Analysis: This category examines the abilities of the source code parser.

- *Parsable source languages*: Columbus currently handles the C and C++ languages, but it can be easily extended to handle other languages as well. An important special capability of the parser is the handling of templates and their instantiation at source level (see Section 3).
- *Project definition types/ease of project definition*: Columbus represents the projects logically in a project-tree with an arbitrary number of subfolders. It is able to handle huge projects, importing MS Visual C++ projects. Handling of several languages in the same project and fine-grained property settings are also possible.
- *Incremental parsing*: Columbus is capable for incremental parsing on project-level (i.e. only the modified files will be re-parsed), which feature is especially useful when extracting from large projects. It uses the precompiled headers technique for speeding-up the extraction.
- *Fault tolerant parser*: The parser is fault-tolerant. It has the ability to parse incomplete and syntactically incorrect source code.
- *Parse speed*: The parser is fast (e.g. it parses 3 million lines of code in about 4 minutes on a PII-400 machine).

Representation: Representations can be divided into textual and graphical reports. Columbus supports both with its built-in diagram viewer and exporter plug-ins.

- *Speed of generation*: Textual reports (e.g. HTML) and the diagrams inside Columbus are created in seconds. The graphical reports take a longer (but acceptable) time mainly due to the COM interface, which is used for the communication between Columbus and the target application (e.g. TED, Rational Rose [15]).

[1] Formerly called TDE (Telecom Design Environment)

- *Filters, scopes, grouping*: Filtering is especially useful when extracting large projects. Columbus offers a three-stage filtering mechanism: filtering by input files, filtering according to scopes and filtering using class dependencies. In addition, classes can be individually selected/deselected on the displayed class diagram.
- *Sorting*: The HTML report generated by Columbus is highly structured (represented in tree-views according to scopes or inheritance).
- *Layout algorithms*: An algorithm is used, which gives priority to inheritance dependencies [10].
- *View editable*: Because Columbus's main task is to extract and filter information, the possibility of editing is passed to the target application (e.g. TED).

Editing/browsing:

- *Integrated text editor/browser*: Columbus has a built-in text editor for viewing/editing the input files and a class browser in a form of a tree-view.
- *External editor/browser*: Columbus supports OLE technology, thus it can use any editor, which is an OLE server.

General capabilities:

- *Toolset extensibility*: Columbus is easily extendible due to its plug-in architecture. New language extractors and representation/output formats can be easily added to the system using the plug-in API.
- *Storing capabilities*: During the analysis Columbus stores the extracted information for every source file in a separate binary file.
- *Output capabilities*: After the extraction process the extracted data can be exported into various formats (TDE Mermaid, TED, Rose, MS Jet, HTML, ASCII).

In the next section we will describe the subject system in detail. One of the most important modules of Columbus is the C/C++ analyzer, which is described in Section 3. In Section 4 we show how Columbus helps in creating comprehensible diagrams. Section 5 presents experiments made with our tool, while Section 6 discusses some tools with similar objectives. Finally, in Section 7 we draw some conclusions and outline further work.

2 COLUMBUS

In this section we will describe the subject system in detail. Figure 1 shows a typical snapshot of a Columbus session.

The main motivation for developing the Columbus system was to create such a tool, which implements a general framework for combining a number of reverse engineering tasks and to provide a common interface for them. Thus, Columbus is a framework, 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

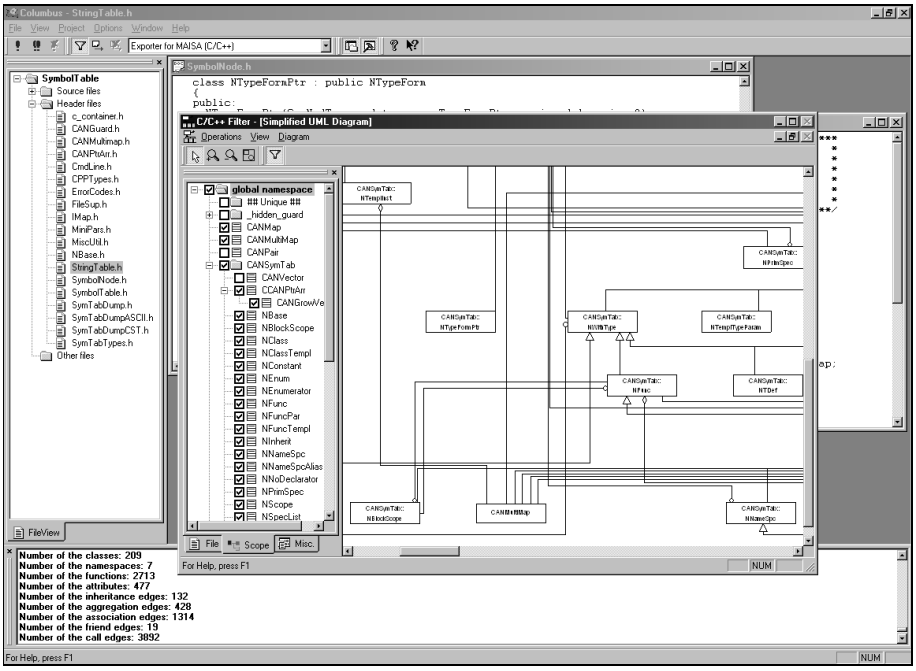

Fig. 1. The User Interface of Columbus is very similar to IDEs

present as basic parts of Columbus, while the system can be extended for other reverse engineering requirements as well. This way we get a really versatile and an easily extendible tool for reverse engineering.

2.1 Overview of the Columbus System

The basic operation of Columbus is performed by the use of three types of plug-ins (in form of MS Windows DLLs). These are the following:

- *Extractor plug-ins* (currently an extractor for C/C++) – The task of an extractor plug-in is to properly analyze a given input source file and to create a file, which contains the extracted information.
- *Linker plug-ins* – The task of a linker plug-in is to build up (in the memory) the complete merged internal representation of the project. This process is carried out based on the files created by the extractor plug-in. This plug-in is responsible also for filtering the merged data in order to produce a more clear-cut internal representation for exporting.
- *Exporter plug-ins* – The task of an exporter plug-in is to export the internal representation built up and filtered by the linker plug-in into a given output format. (The currently available exporters are for: TDE Mermaid 2.2, TED 1.0, Rational Rose, Microsoft Jet Database, HTML, XML and ASCII.)

Beside the delivered 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*.

2.2 Columbus Projects

The extraction process is based on a Columbus project. A project stores the input files (and their settings: precompiled header, preprocessing, output directories, message level, etc.) displayed in a tree-view, which represents a real software-system. The project can *simultaneously* contain source files of different programming languages. Non-source code files can be added to the project as well (e.g. documents, spreadsheets), which are displayed by Columbus using OLE technology.

2.3 The Extraction Process

The complete extraction process in Columbus can be seen in Figure 2.

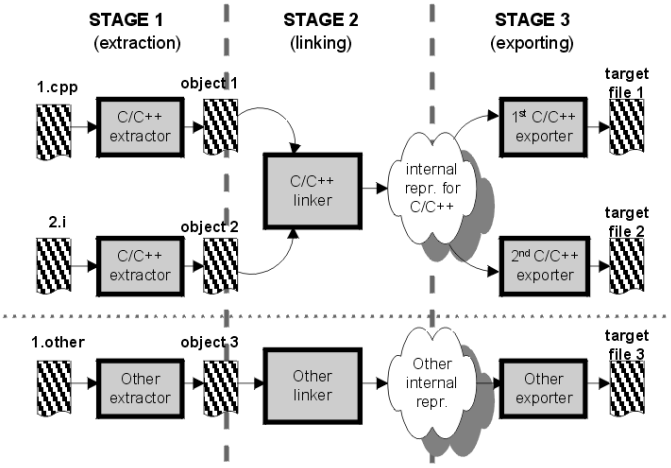

Fig. 2. The extraction process

The whole process is very similar to compiler systems. The first stage of the extraction process is the *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 after selecting the desired export format the *exporting* is performed. The exporting is usually based on a filtered internal representation. Filtering is discussed in detail in Section 4.

All stages of the extraction process can be influenced by setting various plug-in specific options. An important advantage of the Columbus system is that it can *incrementally* perform all of the above described steps, i.e. if the partial results of the certain stages are available and the input of the stage has not been changed, the partial results will not be recreated.

3 CAN – THE C/C++ ANALYZER

The parsing of the input source codes 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++ sources. This allows that it can be *integrated* into the user's makefiles and other configuration files by which it facilitates its 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 asset of CAN is probably the *handling of templates* and their *instantiation at source level*, which is accomplished using a *two-pass technique* for analysis. This way a separate analyzer belongs to both passes, which recognize different things from their inputs. As the task of the first pass is only to recognize the language constructs in connection with the templates, the analyzer of this pass ignores everything else (like a *"fuzzy" parser*). The first pass will be described in more detail in Section 3.1. The second pass performs the complete analysis of the source code and creates its internal representation. So the analyzer of this pass is a complete C++ analyzer. The language description of C++ implemented in the analyzer covers the ISO/IEC C++ standard of 1998 [13]. Furthermore, this grammar is extended by the Microsoft extensions used in Microsoft Visual C++ 6.0.

The information collected by CAN comprises the UML Class Model including C++ templates (definitions, specializations and instantiations) and the call graph. CAN supports the *precompiled headers* technique as well that is widely used by compiler systems in order to decrease compilation time. This technique is efficient especially in case of *large projects*. The parser is *fault-tolerant* (it has the ability to parse incomplete, syntactically incorrect source code), which means that it can continue the analysis from the next parsable statement after the error.

3.1 Handling of Templates

It has been already mentioned earlier that CAN carries out the analysis of the input in two passes. The task of the first pass is to prepare the original input (and to create a temporary file) for the second pass by removing every template

(definition and use) from it and in the same time generating the used template instantiations at source code level. This means, that the second pass is given such a code, which contains only ordinary classes.

One reason for using the above-described technique is that it is rather difficult for the second pass to analyze template definitions because within the definition the formal template parameters behave as regular types or variables and in case of complex structures this causes difficulties to the parser. This means that the kind of symbols created based on formal parameters (e.g. type, function, variable) cannot be determined and this can cause ambiguities in the parser. The other reason is that if once the template instantiations are created, it can be extremely useful for the user to see at source code level what will in fact be instantiated in case of individual template uses.

The basic technique will be presented on a small example shown below. Suppose that the input file for CAN contains the following code:

```
template <class T>
class A {
  T a;
};

char c;
A<int> var;
```

This code is processed by the first pass. During this operation every information is stored in connection with the templates and a file is created whose contents is shown below:

```
class _CTC20B70D45F2;




char c;
class _CTC20B70D45F2 {
  int a;
};
_CTC20B70D45F2 var;
```

The first pass is divided into two phases. In the first phase (1) the template definitions are removed and (2) every template use (instance) is replaced by a newly generated identifier[2] (which probably does not occur in the user's code). In the second phase the template instantiation is carried out. As it can be seen two things will be created out of the `A<int>` template use: (1) the forward

[2] The generated name is a valid C++ identifier. The first six characters are fixed (_CTC20) and it stands for **C**AN **T**emplate **C**ode Version **2.0**. The other eight characters comprise in fact a 4-byte hexadecimal number that unambiguously identifies the instantiation, which it refers to.

declaration of the instantiated class (from the template `A` with the argument `int`) at the place of the original definition of the template and (2) the definition of the same instantiated class at the *nearest point of instantiation* (which is – as seen from the usage point – the last valid line of the innermost open namespace scope). In the example above this is the line, which immediately precedes the declaration of the `var` variable. The forward declaration of the instantiated class is needed because the use of the template can be in a namespace other than the template definition.

In the instantiated class (or function) the formal template parameters are replaced by the actual arguments taking into consideration the possible default arguments. The above discussed instantiation will be created only if the given template has not been instantiated yet with the given arguments in the "visible" scope.

The newly generated identifiers will be of course replaced with the original templated names in the output.

4 PRODUCING COMPREHENSIBLE DIAGRAMS

The reverse engineered code can produce huge amount of extracted data, which is hard to visualize in a way that offers useful information for the user (the user is interested only in parts of the whole system at a time). Different filtering methods in Columbus can help solving this problem.

There are four options for filtering:

- *Filtering by input source files*: only classes that come from the given input files can be selected.
- *Filtering according to scopes.* Classes or namespaces can be selected individually in a tree-view browser.
- *Filtering using class dependencies* (e.g. aggregation, inheritance), with which the given relations can be selected. An interesting and useful feature is *Diagram Completing* with which we can control the possible elements brought in by the relations transitively controlling this way the completeness of the class diagram (e.g. using this option we can select all derived classes of a given class).
- *Filtering "by hand"*: The classes can be individually selected/deselected on the displayed class diagram customizing it this way.

We demonstrate the filtering on a class diagram created with Columbus by extracting the information from our symbol-table implementation code, which is used by CAN. The full class diagram is shown in Figure 3. After applying the default filter the classes from the standard libraries and the structs, unions and template instances are deselected. On the so filtered class diagram some remaining classes were removed "by hand" that are not strictly part of the symbol-table. The resulting diagram (which corresponds to the framed area in the full diagram) is shown in Figure 4.

For *layout*ing the resulting diagrams an algorithm is used, which gives priority to inheritance relation [10].

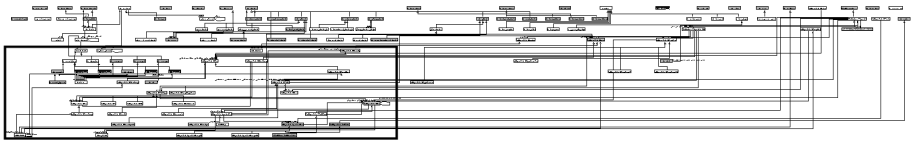

Fig. 3. Full class diagram

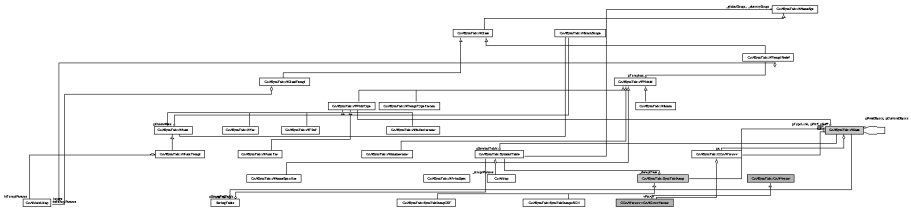

Fig. 4. Filtered class diagram

5 EXPERIMENTS

In this section we demonstrate Columbus's extraction capabilities. The experiments were performed on different C++ projects listed below:

- *Proj.1*: Part of an earlier version of TED[5]. This project was used to demonstrate Columbus's capabilities for *handling sophisticated templates* (part of the Standard Template Library [13]).
- *Proj.2*: IBM Jikes compiler [7]. Using this project we investigated Columbus's capabilities for *handling sophisticated class hierarchies*.
- *Proj.3*: An earlier version of Columbus. This project utilizes the MFC library by Microsoft. This project was used to demonstrate Columbus's capabilities for using *precompiled headers* (and also for handling sophisticated class hierarchies).
- *Proj.4*: A project that implements a graphical library[5]. This is a large C++ project that consist of 226 source files (about 25 million lines of code) and contains only regular classes. Using this project we investigated Columbus's capabilities for *handling real-size, huge projects*.

The following table contains the size information of the projects:

Project infos	Proj.1	Proj.2	Proj.3	Proj.4
No. of files	6	42	8	226
Size (bytes)	5 135 707	17 108 706	22 801 351	200 275 782
LOC[6]	407 448	1 474 765	3 236 641	24 847 279

[5] Proj.1 and Proj.4 were ensured by the Nokia Research Center
[6] LOC: Lines of Code – all files are preprocessed

All tests were performed on an Intel PII-400 machine equipped with 256MB RAM running Windows NT 4.0.

The next table shows the measurement results of the extraction. It contains the extraction time, the extraction time using precompiled headers (only Proj.3 was set up to support this feature) and the memory consumption. We can see that the use of precompiled headers can significantly reduce the extraction time. We can observe that the extraction time and the memory consumption are linear with the size of the input sources. Due to this fact Columbus is able to handle large projects.

Extraction	Proj.1	Proj.2	Proj.3	Proj.4
Extraction time[7]	00:02:20	00:04:17	00:04:43	00:56:59
using precomp. headers	-	-	00:01:39	-
Memory consumption	15 344 K	18 652 K	22 064 K	112 808 K

The table below shows the number of extracted items from the test projects.

Statistics	Proj.1	Proj.2	Proj.3	Proj.4
Classes	1 298	293	1 633	1 307
Namespaces	61	1	1	1
Functions	8 882	12 984	12 207	57 605
Attributes	3 644	1 808	7 194	7 222
Inheritance relations	533	61	425	168
Aggregation relations	222	136	476	480
Association relations	89	493	259	671
Friend relations	48	6	64	41

6 RELATED WORK

In this section we present some tools that have similar objectives as Columbus. We will primarily look at their reverse engineering capabilities.

SNiFF+ [17]: SNiFF+ is more than a reverse engineering tool. It is an open, extensible and scalable program developing environment for C/C++ primarily and for other programming languages (e.g. Java, Fortran) as well. It is available on several platforms and can be easily integrated with version control tools. The reverse engineering part of the tool is based on a fuzzy parser, which is very fast and fault tolerant but in some cases not precise enough (e.g. it silently ignores complex template structures like the STL without any warnings). It produces various diagrams to help program comprehension (e.g. class hierarchy, call graph) but it does not support UML.

WithClass 99 [21]: WithClass 99 is primarily a UML forward engineering tool, but it contains modules to support reverse engineering tasks as well. Similarly to Columbus it is an extensible tool and can be used together with

[7] Time format: hh:mm:ss.

some developing environments (e.g. Visual Studio 97). The parser of the system is able to analyze C/C++ source code but it has difficulties with complex template definitions and handling of namespaces.

Rational Rose 98 [15]: Rational Rose 98 is also primarily a UML forward engineering/modelling tool but it has reverse engineering support for several languages, primarily for C/C++. Similarly to Columbus it uses a separate parser module called Rose C++ Analyzer. It has an intelligent error recovery, which can fix simpler errors during parsing, but it has problems with namespaces and complex templates.

Together/C++ [19]: Together/C++ is a reverse- and UML forward engineering tool in the same time. It can manage source codes and the corresponding diagrams simultaneously and to update diagrams and sources incrementally. However, its parser is slow and does not support real-size, large inputs.

7 CONCLUSION AND FURTHER WORK

In this paper we presented the functionalities of the Columbus toolset with respect to its reverse engineering capabilities.

Columbus supports several reverse engineering tasks (e.g. project handling, data extraction and data representation/visualization with filtering and exporting options). The current version is able to analyze C/C++ projects but due to its flexible architecture it is easy to extend it with other languages as well.

The main features of Columbus can be summarized as follows:

- Effective project handling (capability for importing MS Visual C++ projects, integration into the user's project).
- Powerful C/C++ extraction (fast, fault-tolerant parsing, handling of complex templates, visualizing "hidden" template instances).
- Direct access to the extracted information (*via* its API).
- Creation of comprehensible diagrams (filters, layout).
- Easy-to-use user interface (very similar to IDEs).
- Extensibility (plug-in architecture, user plug-ins *via* its plug-in API).
- Various output formats (Mermaid, TED, Rose, MS Jet, html, XML, ASCII).

In the future we will extend the system for other source languages (e.g. Java) and more output (export-) formats. Further improvements are under development as well, which may be useful for better code understanding (e.g. dependency-graph [6][3][8]). In the future we plan to enhance Columbus so that it supports *architectural reconstruction* of software systems [1] (recognizing design-patterns [9], component interaction, structural information).

References

1. Armstrong, M. N., Trudeau, C. *Evaluating Architectural Extractors.* In Fifth Working Conference on Reverse Engineering. Oct. 12-14, 1998. Honolulu, Hawaii, USA. 30-39.

2. Bellay, B. and Gall, H. *An Evaluation of Reverse Engineering Tool Capabilities.* In Software Maintenance: Research and Practice. 10. 1998, 305-331.
3. Beszédes, Á., Gergely, T., Szabó, Zs. M., Csirik, J. and Gyimóthy, T. *Dynamic Slicing Method for Maintenance of Large C Programs.* In Proc. 5th European Conference on Software Maintenance and Reengineering (CSMR 2001). Lisbon, Portugal, March 14-16, 2001. 105-113.
4. Chikofsky, E. J. and Cross II, J. H. *Reverse engineering and design recovery: A taxonomy.* IEEE Software 7, 1. Jan. 1990. 13-17.
5. *Columbus Setup and User's Guide.* Version 2.5, ©1998-2000 Nokia Research Center.
6. Gyimóthy, T., Beszédes, Á., and Forgács, I. *An Efficient Relevant Slicing Method for Debugging.* In Proc. 7th European Software Engineering Conference (ESEC). Toulouse, France. Sept. 1999. LNCS 1687. 303-321.
7. *IBM Jikes Project.* http://OSS.Software.IBM.Com/developerworks/opensource/jikes
8. Jackson, D. and Rollins, E. J. *A new model of program dependences for reverse engineering.* In Proceedings of the second ACM SIGSOFT symposium on Foundations of software engineering. 1994. 2-10.
9. Keller, R. K., Schauer, R., Robitaille, S. and Pagé, P. *Pattern-Based Reverse-Engineering of Design Components.* 1999. ICSE '99, Los Angeles CA, USA. 226-235.
10. Márton, G. *GraphLayout 1.0: Layout algorithms for software diagrams.* ©1998 Nokia Research Center.
11. Müller, H. *Understanding Software Systems Using Reverse Engineering Technologies: Research and Practice.* In Proc. of the 18th Int. Conf. on Software Engineering. Mar. 1996. Software release v5.4.4.
12. *OMG Unified Modeling Language Specification.* Version 1.3, ©1999 Object Management Group, Inc.
13. *Programming languages – C++.* ISO/IEC 14882:1998(E).
14. Quilici, A. *Reverse engineering of legacy systems: a path toward success.* Proceedings of the 17th international conference on Software engineering. 1995. 333-336.
15. *Rational Rose.* http://www.rational.com/products/rose
16. Riva, C., Przybilski M. and Koskimies, K. *Environment for Software Assessment.* In Workshop on Object-Oriented Architectural Evolution, 13th European Conference on Object-Oriented Programming (ECOOP '99). June 15, 1999. Lisbon, Portugal.
17. *SNiFF+.* http://www.windriver.com/products/html/sniff.html
18. Taivalsaari, A. and Vaaraniemi, S. *TDE: Supporting Geografically Distributed Software Design with Shared, Collaborative Workspaces.* In CAiSE'97 Conference Proceedings, LNCS 1250. Springer Verlag, 1997. 389-408.
19. *Together/C++.* http://www.togethersoft.com
20. *User's Guide to Tde EDitor – TED.* Version 1.0, ©1999 Nokia Research Center. 4.10.1999.
21. *WithClass 99.* http://www.microgold.com