Modelling and Reverse Engineering C++ Source Code

Rudolf Ferenc

Department of Software Engineering
University of Szeged

Supervisor: Dr. Tibor Gyimóthy

November 2004
Szeged, Hungary

A thesis submitted for the degree of doctor of philosophy
of the University of Szeged

University of Szeged
Doctoral School in Mathematics and Computer Science
Ph.D. Program in Informatics

"In science it is not enough to think of an important problem on which to work. It is also necessary to know the means which could be used to investigate the problem."

Leo Szilard

Preface

About the thesis.

This thesis discusses the concepts of *reverse engineering* C++ source code. Reverse engineering is defined as the process of analyzing a software system to create some kind of representation for it at a higher level of abstraction. In this work we are, among other things, engaged in creating a so-called *model* representation of the source code. The format of this model is prescribed by a *schema* whose design is one of the main results of this work. We also present various methods with which the process of reverse engineering C++ source code can be carried out with relative ease.

The second part of the thesis deals with some important utilizations of the generated software model. First, we present two methods for recognizing design pattern instances in C++ code. Second, we describe a work carried out to assess the quality of open source software by predicting its fault-proneness.

Acknowledgements.

First of all, I would like to thank my supervisor, Dr. Tibor Gyimóthy for supporting my work with useful comments and letting me work at an inspiring department, the Department of Software Engineering. I would also thank all my colleagues and friends Árpád Beszédes, Ferenc Magyar, László Vidács, Fedor Szokody, Gábor Lóki, István Siket, Péter Siket, Zsolt Balanyi and László Müller – who all participated in developing and testing the Columbus framework – for striving so hard on this application. I would also like to express my gratitude to David Curley for scrutinizing and correcting this thesis from a linguistic point of view and András Kocsor for his useful advice.

Last, but not least, my heartfelt thanks goes to my wife Györgyi for providing a secure family background during the time spent writing this work.

Rudolf Ferenc, November 2004.

List of Figures

List of Tables

Contents

To the memory of my parents,

Katalin and István.

"All things are difficult before they are easy."
Thomas Fuller

Chapter 1

Introduction

Software systems are rapidly growing and changing, so source code written today becomes legacy code in a very short period of time. This is mainly due to the swiftly changing market requirements and also to the ever-changing new technologies. The always tight deadlines often prevent the developers from properly releasing a product with up-to-date documentation (like design descriptions and source code comments). In such cases the only valid documentation is the source code itself. Some consequences of the above include lots of clones in the source code, dead code and fault-prone code, to mention only a few. *Reengineering* in general seeks to develop methods, techniques and tools to cure these problems and make program comprehension and maintenance easier. The first part of a reengineering process is called *reverse engineering*, which is defined as "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 or at a higher level of abstraction" [16].

The object-oriented paradigm has in recent years become the most popular one for designing and implementing large software systems. By now, many object-oriented systems have reached a state where they can be treated as legacy systems that need to be reengineered. In contrast with older programming languages like COBOL, for instance, reengineering methods for object-oriented programs are not yet fully elaborated. The really large and complex systems like telecom switching software and office suites are generally written in the C++ programming language. This object-oriented language is probably the most complex one and therefore, not surprisingly, it is least supported by reengineering methods and tools. This language offers us reengineers the most exciting challenges and opportunities for research.

To comprehend an unfamiliar software system we need to know many different things about it. We refer to this information as *fact*s about the source code. A fact is, for instance, the size of the code. Another fact is whether a class has base classes. Actually any information that helps us understand unknown source code is called a fact here. It is obvious that collecting facts by hand is only feasible when relatively small source codes

are being investigated. Real-world systems that contain several million lines of source code can be only processed with the help of software tools.

In our approach tool-supported *fact extraction* is an automated process during which the subject system is analyzed on a file by file basis with analyzer tools to identify the source code's various characteristics and their interrelationships and to create some kind of representation of the extracted information. This information can afterwards be used by various reengineering tools like metrics calculators and software visualizers. The format of the outputs of the analyzer tools is unfortunately not standardized, almost every tool having its own format and this can lead to interoperability problems. Every application that would like to use the information gathered by other tools has to implement different convertors to gain access to the data.

This problematic situation was recognized by researchers and significant effort has been made to tackle this problem. One of the fruits of this grand effort is GXL [39] (Graph eXchange Language), which is a fine example of what can be achieved when we focus our energies. But using GXL itself is not enough. It offers a common medium for exchanging graphs (i.e. nodes and edges) with the help of XML, but does not describe how to represent various programming language-specific entities like C++ classes and functions. Researchers previously tackled this problem as well (e.g. [18–21; 30; 38; 47]), and different solutions were proposed but none of these is widely accepted and used. Without a common standard format (schema), smooth data exchange among different C++ reengineering tools is hard to achieve. By *schema* we mean a description of the form of the data in terms of a set of entities with attributes and relationships. A *schema instance* (in other words, a *model*) is an embodiment of the schema which models a concrete software system. This concept is analogous to databases which also have a schema (usually described by E-R diagrams) that is distinct from the concrete instance data (data records). In this work we present among other things a schema for the C++ language which satisfies some important requirements of an exchange format. It mirrors the low-level structure of the code, as well as higher level semantic information (e.g. name resolution and the semantics of types).

Plenty of work has been done in the field of re- and reverse engineering, which make use of the results of fact extraction. These include code measurements (different kinds of metrics), visualization, documentation and code comprehension. But relatively few papers deal with the actual process of fact extraction from C++ source code. We present methods with which automatic fact extraction can be achieved from real-world software systems. We developed a framework called *Columbus* [22–26; 28; 29] for supporting these methods and reverse engineering in general. The framework also takes care of fact representation, filtering and conversion to various formats to achieve tool interoperability. It is now used for research and education in many academic institutions around the world.

Going one step further in abstracting an analyzed software system we developed methods for recognizing design patters in our schema instances; and, this way, in the source code as well. Design patterns abstract practical solutions for frequently occurring design problems to an object-oriented format and are the most natural means when recovering the architectural design and the underlying design decisions from the software code.

To demonstrate that our methods could be used in practice we analyzed the source code of several versions of the popular internet suite *Mozilla* [50; 56] and calculated various metrics from it. We used these metrics to predict the fault-proneness of the source code. We also compared the metrics of seven versions of Mozilla and showed how the predicted fault-proneness of the software changed during its development cycle.

In the next section we will provide a concise summary of this thesis according to its structure then, in Section 1.2, we will summarize the results of the author.

1.1 Summary by chapters

The present thesis consists of two main parts. The first part (Chapters 3 and 4) discusses source code modelling and the fact extraction methodology, while the second part (Chapters 5 and 6) deals with the applications of these results in various reverse engineering tasks.

Part I: Fact representation and extraction

In the second chapter we first provide an overview of the state-of-the-art of reverse engineering C++ source code. In addition, we define the terms and expressions used in this thesis and we outline tools, schemas and approaches having similar objectives to ours. Lastly, we state some requirements that we wanted to realize.

The third chapter presents a detailed description of our schema for C++. The schema is introduced with standard UML class diagrams. Besides the explanations of the diagrams an example C++ source code and its schema instance are included to help in their comprehension. The chapter also mentions some successful data exchange cases with other tools.

The fourth chapter exploits our fact extraction process. What steps need to be taken to carry out a real-life fact extraction task are described. The chapter also contains the description of the Columbus framework, its visual interface and command line tools that support the process. The chapter also deals with the conversion of the schema instances to various formats to aid data exchange among tools. Several well-known file formats are mentioned for which conversion algorithms are available in the Columbus framework. The chapter also describes different outputs derived from our schema. These are produced by more sophisticated applications, like calculating metrics and mining design patterns.

Part II: Utilization of the extracted facts

Chapter 5 describes two methods for recognizing design patterns in C++ source code. The first method is based on a project carried out in collaboration with the University of Helsinki where we basically integrated Columbus with the design pattern miner tool *Maisa* [52; 55]. This research project showed that our framework had the capacity to convert the extracted facts (which are represented according to our schema) to another format (a PROLOG-like format in this case) to be successfully used by another tool (Maisa). The second method is a sophisticated design pattern miner algorithm which looks for occurrences of design pattern instances directly in our C++ schema instances. This is a complex, parameterizable, fast graph matching algorithm which can be used on real-world software. The design patterns that are searched for can be described with the help of our new XML-based Design Pattern Markup Language (DPML).

The sixth chapter presents a real case study in reverse engineering for predicting the fault-proneness of the well-known open source web and e-mail suite *Mozilla* [50; 56].

The source code of Mozilla was analyzed using our process and tools, and then object-oriented metrics were calculated from the built-up schema instances. Afterwards these metrics were analyzed to predict the fault-proneness of the software.

Chapter 7 draws some conclusions and provides a short summary of the results achieved. In this chapter we also evaluate our framework according to the requirements given in the second chapter.

Lastly, we round off with appendices that contain the descriptions of our C++ Markup Language (CPPML) and Design Pattern Markup Language (DPML), followed by a brief summary of the principal results of the thesis in English and Hungarian.

1.2 Summary by results

The main contributions of this work can be summarized as follows. Most importantly, we define in detail a process for automatic tool-supported fact extraction and presentation and a schema for representing C++ source code entities and relations. We also present our reverse engineering framework that most ably supports our process. Furthermore, we show how tool interoperability can be achieved by using our schema instance conversions. To demonstrate the operability of our process and framework and the usability of our schema, we introduce two large applications. First, we present methods for recognizing design patterns in C++ source code; and second, we analyze the fault-proneness of open source software. Both applications are built on our general framework for reverse engineering.

We state four main results in the thesis and the contributions of the author are specified in the listed results. As the thesis consists of two main parts, the results are also separated into two parts.

Part I: Fact representation and extraction

The results of the first part of the thesis include the schema for the C++ language and the process of fact extraction and presentation. These are elaborated on in Chapters 3 and 4, respectively.

1. **Schema for the C++ programming language.**
   This work was motivated by the observation that successful data exchange among reengineering tools is of crucial importance. This requires a common format so that the various tools (like front ends, metrics tools and clone detectors) can "talk" to each other. The author designed a schema, called the *Columbus Schema for C++* [21; 26; 30], which captures information about source code written in the C++ programming language. The schema is modular, so it offers further flexibility for its extension or modification. The schema is fine-grained, representing practically every relevant fact about the source code so that logically equivalent source code can be generated from its instances. This schema seeks to fill a gap that was present in the literature of reengineering science since the design of the C++ language. Nobody has designed a C++ schema before in such detail as the author has. The reason for this is probably the extreme complexity of the C++ language. This first result of the thesis includes, in addition to the design of the schema, its implementation by the author (which is used by our C++ analyzer tool as well) and algorithms for name resolution, type-checking, serialization, class diagram generation and call graph creation, to name only a few. Note, however, that the building of schema instances during source code parsing is not the work of the author.

2. **C++ fact extraction process and framework.**
   Extracting facts from a small program is simple enough and can be done even by hand. The real challenge is to analyze real-world software systems that consist of several million lines of code. The literature lacked methods and the community did not have tools with which such a complex task could be efficiently performed. The author presents a process [29] that lists five key steps which have to be done to successfully carry out a fact extraction task from C++ source code. The process deals with important points such as handling configurations, linking the schema instances, filtering the data obtained and converting it to other formats to facilitate data exchange. We also present the Columbus reverse engineering framework [26] in detail, which readily supports the process. The framework is widely used at universities across the world, and so far over 600 downloads of the framework have been registered. The author designed and implemented the following parts of the framework (see Section 4.2):
   Columbus REE, *C++ linker plug-in*, *CPPML/GXL/Maisa exporter plug-ins*, *Columbus IDE Add-ins*, *CANLink*, *CANFilter* and the *CANGccWrapper* toolset, not to mention the following conversion algorithms: *CPPML – C++ Markup Language* (including the design of the language), *GXL – Graph eXchange Language* and *Maisa* (the algorithms were implemented within the *CAN2Cppml*, *CAN2Gxl* and *CAN2Maisa* tools). The author also participated in the design of the *Design Pattern Miner* module. Note though that the C++ Analyzer tools (*CAN* and *CANPP*) and the rest of the conversion algorithms and derived outputs are not the work of the author.

Part II: Utilization of the extracted facts

The topics of the second part of the thesis are two applications of the extracted facts described in the first part. These are presented in Chapters 5 and 6 of the thesis, respectively.

3. **Design pattern recognition in C++ source code.**
   Existing reverse engineering tools and methods produce a wide variety of abstract software representations. A natural strategy of abstracting object-oriented programs is to represent them as a set of UML diagrams [53]. While the automatic generation of UML diagrams from software code is supported by a number of reverse engineering tools, recognizing *design patterns* [34] is, currently, almost totally without advanced tool support. Design patterns are the most natural and useful assets when recovering the architectural design and the underlying design decisions from the software code. The third main result of the thesis offers two methods for discovering design pattern instances in C++ source code. First, we present a method [27] and toolset for recognizing design patterns with the integration of Columbus and Maisa [52; 55]. The method combines the fact extraction

$\mathcal{N}o.$	[30]	[21]	[26]	[27]	[3]	[29]
1.	•	•	•			
2.			•			•
3.				•	•	
4.						•

Table 1.1: The relation between the thesis topics and the corresponding publications.

capabilities of the Columbus framework with the pattern mining ability of Maisa. The author implemented the schema instance converter algorithm which produces data in the input format of Maisa's. Second, we present a new solution to the problem of pattern detection with a sophisticated, parameterizable, fast graph matching algorithm that recognizes design patterns in our schema instances [3]. It includes the detection of call delegations, object creations and operation redefinitions. These are the elements that identify pattern occurrences more precisely. The pattern descriptions are stored in our new XML-based format, the *Design Pattern Markup Language* (*DPML*). This gives the user the freedom to modify the patterns, adapt them to his or her own needs, or create new pattern descriptions. The author designed the DPML language and he participated in working out the theory of the algorithm (the concrete implementation was not the work of the author).

4. **Analysis of the fault-proneness of open source software.**
   Open source software systems are becoming evermore important these days. Many large companies are investing in open source projects and lots of them are also using this kind of software in their own work. As a consequence, many of these projects are being developed rapidly and quickly become very large. But because open source software is often developed by volunteers in their spare time, the quality and reliability of the code may be uncertain. Various kinds of code measurements can be quite helpful in obtaining information about the quality and fault-proneness of the code. We calculated the object-oriented metrics validated in [5], [7], [8] and [9] for fault-proneness detection from the source code of the open source internet suite *Mozilla* [50; 56] with the help of our framework. We then compared our results with those presented in [5]. One of our aims was to supplement their work with metrics obtained from a real-world software system. We also compared the metrics of seven versions of Mozilla to see how the predicted fault-proneness of the software changed during its development cycle. The author performed the fact extraction process with compiler wrapping and examined Mozilla's changes [29].

Lastly, Table 1.1 summarizes which publications cover which results of the thesis.

"Climb mountains to see lowlands."
Chinese Proverb

Chapter 2

Reverse engineering

This thesis deals with the modelling of and reverse engineering methods of C++ source code. The problem is not new. Other re- and reverse engineering tools exist as well and they all work with some kind of model of the software. This chapter presents the state-of-the-art in re- and reverse engineering schemas and tools. It starts by defining the terms used, then it discusses re- and reverse engineering and gives an overview of existing schemas and tools.

2.1 Terminology

This section presents definitions for re- and reverse engineering. It is based in part on the taxonomy given by Chikofsky and Cross [16]. We begin with a formal definition of reengineering:

> "*Reengineering* is the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" [16].

This means that reengineering consists of three main activities, namely the examination, the modification and the rearrangement of a subject system. These activities are covered by the concepts of reverse engineering and forward engineering:

> "*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 the system in another form or at a higher level of abstraction" [16].

> "*Forward engineering* is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system" [16].

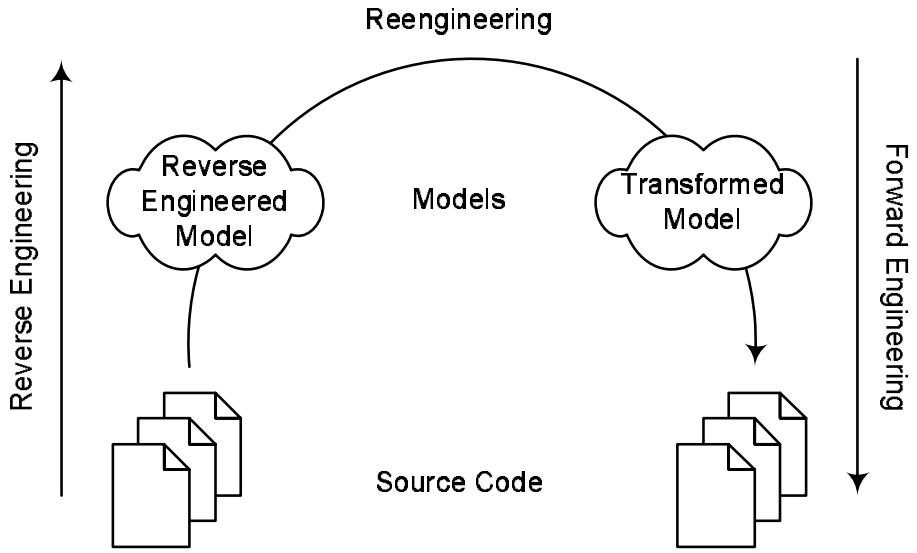


Figure 2.1: The reengineering process.

Figure 2.1 illustrates these concepts [12]. The general aim of reverse engineering is to create models – higher level views – of an existing software system. The principal goals are to comprehend a system, create documentation for it or detect problems. Conversely, the general aim of forward engineering is to create concrete implementations from the high-level models. Reengineering is a combination of the two, namely it transforms concrete implementations to other concrete implementations.

Here we are engaged only in reverse engineering, more precisely in the reverse engineering of object-oriented systems written in the C++ programming language. So we will proceed by defining some concepts related to our concrete research topic.

In reverse engineering, the main objective is to extract information about the static structure and the dynamic behavior of the code and then give it some abstract representation so as to make it easier to explore the essential aspects of the system (by ignoring insignificant implementation details). In the ideal case the low-level code would be reverse engineered back to its original design, or at least to a form that might have been the intent of the software designers.

*Grammar*s are used to define the syntactic structure of a programming language. For a given sentence that belongs to the language, the grammar induces a certain derivation (or several if the grammar is ambiguous), characterized by a *parse tree*. This parse tree is determined by the underlying grammar and may contain unnecessary artifacts like chain rule derivations.

The purpose of an *abstract syntax tree (AST)* is to describe the syntactic decomposition of the represented program, which is basically a tree-nesting structure, but without any unnecessary detail. What constitutes an 'unnecessary detail' depends upon the intended use of the abstract syntax tree. Compilers, for instance, are generally not

interested in brackets used to specify associativity within expressions as this information is shown in the tree nesting structure of these expressions. On the other hand, source-to-source transformation tools have to reproduce the code as true to the original as possible and therefore need to retain information about brackets.

During the semantic analysis phase of front ends the ASTs are usually decorated with some additional information like the resolution of names. These decorated ASTs are referred to as *abstract semantic graphs (ASG)*. The ASG is regarded as a *model* of the source code.

By *schema*, we mean a description of the form of the model (ASG) in terms of a set of entities with attributes and relationships that prescribe the form of the graph. Actually, the schema means the *metamodel* of the source code (the terms "schema" and "metamodel" really refer to the same concept, and both are used in the literature). A *standard schema* is a schema that tool builders have agreed upon to facilitate data exchange between tools.

2.2 Tools

There are surveys available [2; 60] which discuss and assess existing reengineering tools, but not all of them are dealt with. We will also focus only on those tools that are of interest to us here in the thesis.

Most of the tools store information about a software system in a repository. Some contain parsers that extract information from source code, and model importers that load models stored using an exchange format. The properties of the repository are heavily influenced by the schema that describes what information the repository contains. The schema not only determines if the right information is available to perform the intended reengineering tasks, but it also influences scalability, extendibility and information exchange. Section 2.3 outlines some of the schemas in existing tools.

Several groups of tools exist nowadays. There are the general visualizers, not necessarily intended for software reengineering. Then there are tools that are especially designed for certain programming languages.

The first group of tools we discuss are the generic visualizers. They are typically based on simple generic metaschemas to be able to easily handle different kinds of information. The *rigi* program [51; 57] supports reverse engineering by providing a scriptable tool with grouping and graph layout support. It is based on a graph metaschema, enabling it to easily visualize any entity-relationship model. The actual schema can be constructed by the user, applying his/her domain knowledge. For storing and exchanging models rigi provides its own format, the Rigi Standard Format (RSF). Similar to rigi is *SHriMP* [63], a tool to visually browse and explore complex graph-based information spaces. Exploring large programs is only one of their many possible applications.

Other tools are intended for specific languages. Consequently, their schemas are highly language dependent. These tools are presented in the following.

Datrix

The Datrix team, a part of Bell Canada's Quality Engineering and Research group, implemented source code assessment tools with the goal of evaluating the maintainability and evolution of software products [48]. *Datrix* is a command line analyzer tool that extracts information from C/C++/Java source code files according to the Datrix ASG Model [6]. It creates separate output files for individual compilation units. These output files can be in TA (Tuple-Attribute Language) [40] form or VCG [59] form. To handle C/C++ sources, Datrix also has a preprocessor utility. Unfortunately the project at Bell Canada came to an end in 2000 before all planned tools could be implemented. For example, the Datrix toolset lacks a linker for merging the extracted ASGs, so viewing the whole system in its completeness is not possible.

CPPX

CPPX [18] is a free, open source parser and fact extractor for C++ developed at the University of Waterloo. It relies on the preprocessing, parsing, and semantic analysis of the GNU g++ compiler, and produces a graph based on the Datrix ASG Model [6], in either GXL [39], TA, or VCG format. Because it relies on the output of a real C++ compiler, it produces precise data about the analyzed system, but it has the drawback that it cannot handle other C++ language dialects.

TkSee/SN

TkSee/SN is a tool that parses C++ using Source-Navigator as a front end. It creates a graph according to DMM (Dagstuhl Middle Metamodel) [46] represented in GXL.

Source-Navigator

Source-Navigator [61] is an Integrated Development Environment (IDE) that also includes support for code comprehension. Source-Navigator parsers scan through source code, extracting facts from C, C++, Java, COBOL and other programs and then use this information to build up a project database. Source-Navigator's graphical browsing tools utilize this database to query symbols (such as functions and classes) and the relationships among them. Its database has a general format that is suitable for storing high-level information about projects written in several programming languages. For this reason, it provides no explicit schema. However, it provides an API for accessing the extracted information stored in binary files whose form is similar to tables in relational databases.

Acacia

Acacia, the C++ Information Abstraction System, is a collection of analysis and reverse engineering tools for C++: CCia creates a program database from C++ source files to store information about C++ program entities (functions, variables, types, macros and files) and their relationships. The database can be queried and navigated graphically using CIAO which is included as part of the distribution. For more traditional database queries and primitive access to the database information, Acacia contains a set of query tools built on top of cql. The tool supports reachability analysis and dead code detection for C++ applications [14].

2.3 Schemas

There are several (mainly research) groups that have created schemas to represent software. Only a few of the schemas, however, have explicit descriptions of what information is represented. We introduce some of those schemas in this section.

Datrix ASG Model

Datrix [48] is a source code analysis tool developed at Bell Canada. The format of its internal representation is called the Datrix ASG Model [6; 38]. The ASG (abstract semantic graph) represents an AST (abstract syntax tree) with additional semantic information like the identifier's scope and variable's type. The main aims of the Datrix ASG model are completeness (all kinds of reverse engineering analyses should be 'doable' on an ASG) and language independence (the model should be the same for C++, Java and similar languages). This language independence is, however, restricted to C++-like languages.

TA and TA++ – Tuple-Attribute Language

TA [40] is a language for representing graphs. It defines a simple format to describe graphs and a basic schema for describing program entities such as procedures and variables and relationships like calls and references. TA++ [45] is an extension to TA that describes a schema for program-entity level information. Its aim is to provide a representation of high-level architectural information about very large software systems.

GXL – Graph eXchange Language

GXL (as a successor of TA) is designed to be a standard exchange format for graphs [39]. This exchange format offers an adaptable and flexible means for supporting interoperability between graph-based tools. In particular, GXL was developed to enable interop-

erability between software reengineering tools and components, such as fact extractors, analyzers and visualizers. GXL allows software reengineers to combine single-purpose tools into a powerful reengineering workbench.

There are two innovative features in GXL that make it well-suited as an exchange format for software data. First, the conceptual data model is a typed, attributed, directed graph. Second, it can be used to represent instance data as well as schemas for describing the structure of the data. Moreover, the schema can be explicitly stated along with instance data.

DMM – Dagstuhl Middle Metamodel

The Dagstuhl Middle Metamodel (DMM) [46] is an extendible schema for static models of software. It is a middle-level schema since it captures program level entities and their relationships, rather than a full abstract semantic graph (lower level) or architectural abstractions (higher level). DMM can be used to represent models based on information extracted from software written in most common object-oriented and procedural languages.

Bauhaus Resource Graph

The Bauhaus Resource Graph models source code by providing higher level information such as call, type and use relations [17]. It models constructs of procedural programming languages for architecture recovery. Besides the schema, it defines a compact graph-based exchange format.

Acacia

Acacia offers a C++ schema for reachability analysis and dead code detection [14]. The schema models software at the program entity level. The schema is specifically designed for the C++ language, hence it contains C++ specifics such as friend relationships, class and function templates, macros and C++ specific primitive types. It also deals with nested classes and their references in a consistent way.

2.4 Discussion

One of the main topics of this thesis is concerned with the modelling of the source code of object-oriented software for the purpose of re- and reverse engineering. Looking at the state-of-the-art of reverse engineering from this perspective we can conclude that *in most cases there are no detailed schema descriptions*.

The repository and its schema are an essential part of a reengineering framework. However, although many tools exist, with most of them the schema is not explicitly

documented. Consequently, every time a tool is developed the same work needs to be done from scratch, which is obviously wasteful and needless. Furthermore, multiple (undocumented) schemas hinder data exchange and interoperability among tools.

There are of course a few exceptions. Datrix clearly describes what it models, namely abstract semantic graphs of C++ and Java programs. Basically, the program entities are modelled in great detail (with some minor deficiencies). Only TA++ and DMM are general schemas for reverse engineering targeted for multiple languages. They model high-level constructs and have well-defined exchange formats.

2.5 Requirements

In this section we list the requirements we wanted to fulfill in our work. We distinguish three main requirements for C++ fact extractor frameworks which are needed to successfully perform a fact extraction task and briefly describe how we accomplished them.

1. **Analysis of source files**. When speaking about fact extraction one of the most important requirement is, of course, the ability to analyze the source code. To achieve this goal different tools are needed which read the source code and extract the information needed from it.

    - *Extracting preprocessor-related facts*. Macro processing, conditional compilation and file inclusion make the C/C++ preprocessor a powerful tool for programmers. However, having program code with lots of preprocessor directives makes program understanding harder. The problem is that the code that the programmer sees and the preprocessed code that the compiler gets can be quite different. We developed a preprocessor (see Section 4.2.4) that is able to produce the same output as the usual preprocessors but it also collects facts (related to preprocessing) which are represented according to our preprocessor schema.
    - *Extracting C++ language-related facts*. C++ is one of the most complex and sophisticated programming languages ever designed. And what is more, several different dialects of this language now exist. This makes developing a C++ analyzer a real challenge. We developed a tool (see Section 4.2.4) that is able to process C++ code written in several dialects (ANSI, Microsoft, Borland, GNU). The tool collects practically every important fact from the source code so that a logically equivalent code can be generated from them. The collected facts are represented according to our C++ schema (see Section 3.1).

2. **Capability for handling large projects**. It is essential for a reverse engineer to possess methods and tools with which real-world software systems (having perhaps several million lines of code) can be relatively easily and efficiently handled. Besides these methods we will also point out some important issues connected to them.

   - *Acquiring project/configuration information*. Even in the case of small programs, the source code is usually split into a number of files. Furthermore, these files are arranged into folders and subfolders. Compiling such a system by hand with command line compilers is cumbersome, so different techniques were developed to support this. The two most commonly used approaches are: (a) the *make* tool, which is a command line solution and (b) different *IDE*s – Integrated Development Environments – which are visual solutions. In the first case the information we are looking for is stored in so-called *makefile*s, while in the second case it is contained in different *project files*, whose formats are unfortunately different in the case of different IDEs. We introduce a so-called *compiler wrapping* technique for using makefile information (see Sections 4.2.3 and 6.2) and two different approaches for handling IDE project files: IDE integration (see Section 4.2.2) and project file import (see Section 4.2.1).
   - *Filtering*. In the case of large projects, code analyzer tools can produce huge amounts of extracted data, which is hard to present in a way that provides useful information to the user (he/she is usually interested only in parts of the whole system at a given time, like for instance a subsystem or only an individual class). We developed different filtering methods that help in solving this problem (see Section 4.2.5).
   - *Performance*. The front end should be as fast as possible because of the generally large amount of source code that needs to be processed. (This issue is probably the least important, because in a typical reengineering task it must be done only once at the beginning of the project.)

3. **Connectivity with other tools**. The extracted information should be presented in a form that allows other tools to use this data in subsequent operations.

   - *Fact representation*. Every tool in the fact extractor system should 'speak' the same language, that is, they should be able to seamlessly exchange the collected facts. In other words, the extracted facts should conform to a common schema. Moreover, this schema should be modular to allow its easy extension and modification. We designed two schemas that prescribe the form of the facts: a C/C++ preprocessing schema and a schema for the C++ language itself (see Section 3.1).
   - *Schema instance conversions*. The ideal solution would be that every C++ re- and reverse engineering tool in the world uses the same schema for repre-

senting their data. This, unfortunately, is not the case, so the data (schema instances) must be converted to different formats. To make this task easier, the extracted data should be accessible though an API. We can convert the extracted facts to several well-known formats to make the access to them easier in other re- and reverse engineering tools (see Section 4.2.6).

Part I

Fact representation and extraction

"One line alone has no meaning, a second one is needed to give it expression."

Eugene Delacroix

Chapter 3

Fact representation

From the discussion in Chapter 1 it can be concluded that schemas play a very important role in the process of fact extraction. They define the central repository of the whole process from where the facts can be accessed with the help of various conversion algorithms.

We designed two schemas that prescribe the form of the facts: the Columbus Schema for C/C++ Preprocessing [66] (for preprocessing-related facts) and the Columbus Schema for C++ [21; 26; 30] (for the C++ language itself). The descriptions of the two schemas are given using standard UML class diagrams [53], which permits their simple implementation and easy physical representation (e.g. using GXL [37; 39]). Despite the fact that it is not suitable for formal descriptions, we chose UML because it is the de facto standard in object-oriented design so the schema can be relatively easily comprehended by the users.

This thesis does not deal with preprocessing-related facts (the author was only partly involved in its design), so we will concentrate on the Columbus Schema for C++. This chapter describes it in detail.

3.1 The Columbus Schema for C++

The Columbus Schema for C++ satisfies some important requirements of an exchange format. It mirrors the low-level structure of the code as well as higher level semantic information (e.g. name resolution and the semantics of types). Furthermore, the structure of the schema and the standard notation used (UML class diagrams) made its implementation straightforward and, what is even more important, an API was very simple to prepare as well. Hence the Columbus Schema for C++ is a good candidate for exchanging information among tools of various types, e.g. C++ parser front ends and code-rewriters, metrics tools, documentation tools, and even compilers. This is already supported by several studies where the Columbus Schema for C++ has been successfully

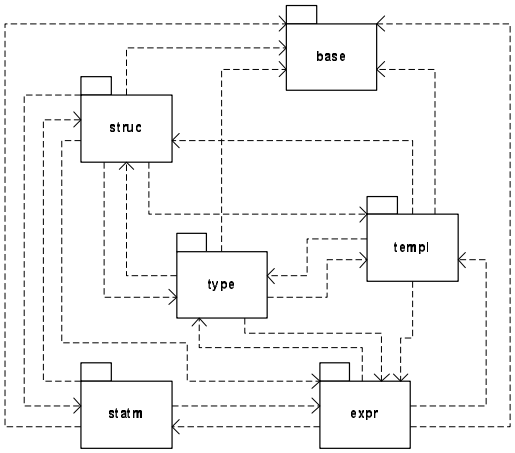


Figure 3.1: Main class diagram.

used for data exchange. Perhaps the most important of these applications is GXL, a new standard for information interchange in re- and reverse engineering [37; 39].

The ISO/IEC C++ standard of 1998 [41; 64] served as the basis for all design decisions. More precisely, the schema models the "clean" C++ language syntax (preprocessed source code); it does not deal with macros or other preprocessor issues (the Columbus Schema for C/C++ Preprocessing deals with them).

The Columbus Schema for C++ is also used as the internal representation of the Columbus framework (see Section 4.2). The schema evolved into its current state in parallel with its implementation and is also used for analysis (e.g. resolving type names and scopes) as well as data exchange.

In the following we will present our schema in detail, followed by an example.

3.1.1 The structure of the schema

Owing to the great complexity of the C++ language, we decided to modularize our schema in a similar way to that proposed in the discussion part of [30] (see Figure 3.1). This also opens up the possibility for its extension and modification. We divided the schema into six packages. These are the following:

- *base*: the base package contains the base classes and data types for the remaining parts of the schema.
- *struc*: this package models the main program entities according to their scoping structure, e.g. objects, functions and classes.
- *type*: the classes in this package are used to represent the types of the entities in the *struc*, *templ* and *expr* packages.

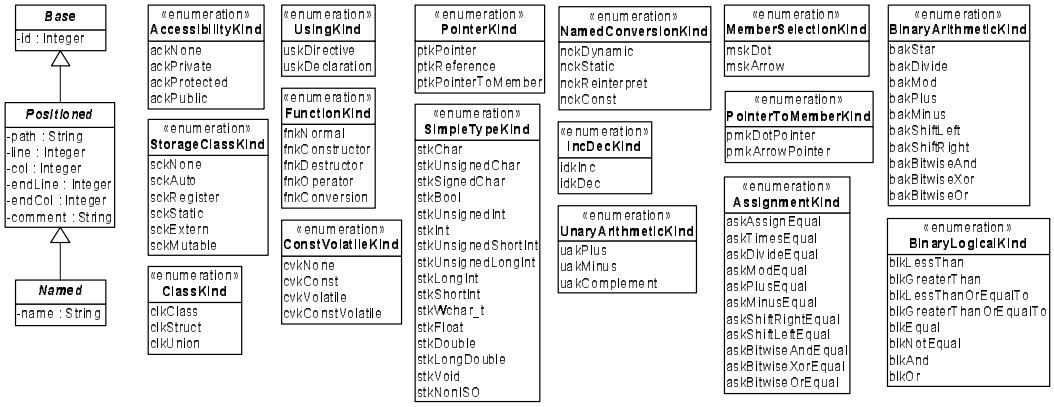


Figure 3.2: Class diagram of the ***base*** package.

- *templ*: the package covers the representation of template parameter and argument lists, and is used by the *struc type* and *expr* packages.
- *statm*: the package contains classes that model the statements.
- *expr*: the classes in this package represent every kind of expression.

In the following we will present these packages one at a time.

3.1.2 *base* – the base package

The base package (Figure 3.2) contains the abstract class *Base*, which is the base class of all classes in our schema. A singly rooted hierarchy has many advantages (e.g. all classes have a common interface). It has one attribute: the node identifier *id*. The second class in this package is called *Positioned*. This abstract class extends *Base* with location (*path*, *line*, *col*umn, ...) and *comment* information. This class is the base class of all classes that represent source code entities having a position in the code. The third class *Named* extends the previous one with *name* information.

Apart from these classes the package contains various enumerations (*AccessibilityKind*, *StorageClassKind*, *ClassKind*, etc.) used by the schema.

3.1.3 *struc* – the structure package

This package contains classes that model the main program entities and their scoping structure, which are organized around *Member*, the most important child class of *base::Named* (see Figure 3.3). This abstract class is the parent of every kind of entity, which may appear in a scope (we use the term "member" in a more general way than

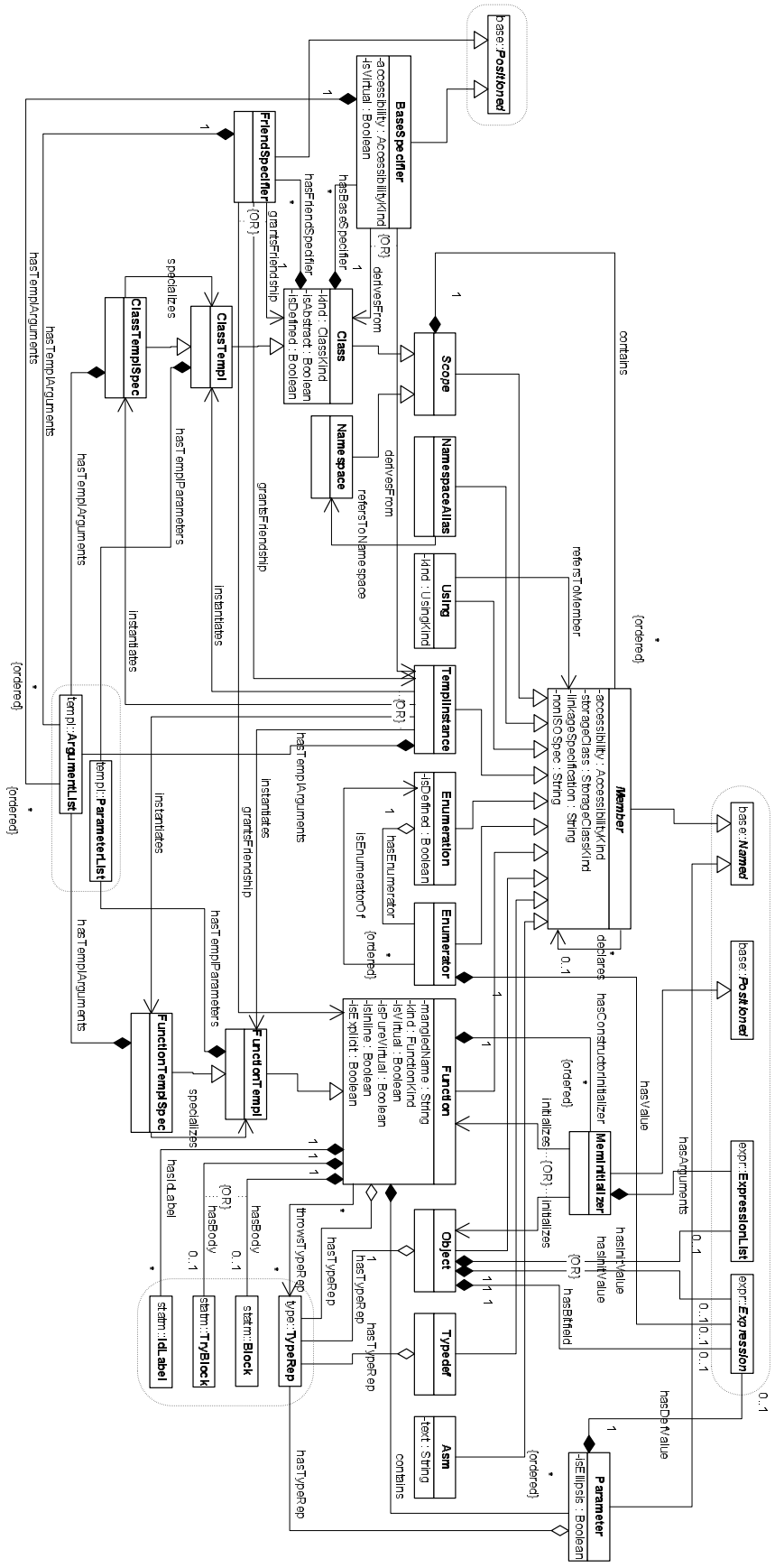


Figure 3.3: Class diagram of the ***struc*** package.

usual). It has the properties called *accessibility* (private, protected, ...), *storageClass* (static, extern, ...), *linkageSpecification* and *nonISOSpec* for capturing any non-ISO specifiers from different language dialects. In addition, member declarations may point to their definition.

The first child of the *Member* class is the abstract class *Scope*, which is a member as well because it can be contained by another scope. The composition from *Scope* to *Member* enables the class to store other members in an ordered way, which is a natural representation of the scope nesting in the C++ language (i.e. it is a composite). This recursive containment along with the other compositions builds the basic *skeleton structure* tree of the modelled system.

The *Namespace* class is used to represent C++ namespaces. In a schema instance there must always exist at least one namespace object. This object is called *"global namespace."* Our schema is essentially designed to be project-oriented, so namespace scopes have priority over file scopes (both cannot be reasonably represented in an ASG at the same time because namespaces can be defined across files/compilation units). However, the original path information is stored in the schema and the files can be recovered from there.

The class named *Class* is similar to *Namespace* because both represent scopes. It has the field *kind* (class, struct or union) and others that say whether it is abstract and defined or not. Additionally, it is composed with two classes that represent its base classes and friends. The first one is the class *BaseSpecifier*, which models the inheritance relationship between two classes or between a class and a template instance. The information about accessibility and virtuality is stored as attributes. The second one is the class *FriendSpecifier*, which stands for the friend relationship between two classes, between a class and a function, or between a class and a template instance. When modelling these two relationships among classes and template classes/functions, these classes are also composed with template argument lists that represent the actual template arguments used for the instantiation of the referenced template. For instance, when deriving class `A` from the template class `C` (which is nested in template class `B`), the base specifier will contain the argument lists `<int>` and `<D,char>` for the C++ code `class A : public B<int>::C<D,char>`.

The classes *Function*, *Object*, *Typedef* and *Parameter* are very similar, so they will be described together. Basically, these are the language entities that have a type. Our schema represents this by aggregations with the *TypeRep* class from the *type* package (see Section 3.1.4 for more on this). Note that using this kind of representation of types makes it possible to store each type representation only once and point to them from multiple nodes. Some of these classes have some custom attributes that are needed for storing special information like function *kind* (constructor, destructor, etc.) or if the parameter is an *ellipsis* or not. *Function*s have additional associations: *throwsTypeRep* models the eventual exception specifications defined by the function, while *hasBody* and *hasIdLabel* are used to model the body (block scope with statements) and the jump

labels of the function. The *hasConstructorInitializer* composition stands for the constructor initializer list (if the function is a constructor, of course). The class called *MemInitializer* models one entity in this list and it points to a *Function* or an *Object* being initialized. The concrete initialization value(s) are stored in an *expr::ExpressionList* (see Section 3.1.7). The schema does not distinguish between representing class attributes and global/local variables (these are all *Object*s). *Object*s also have additional associations: *hasInitValue* for modelling the eventual initialization value and *hasBitField* for capturing its bitfield width (if it represents a bitfield). *Typedef*s are special because they are not "real members" in the usual sense (just type aliases), but syntactically they look almost exactly the same as *Object*s. Note that the class *Parameter* is not a member (it is derived from the class *base::Named*). Instead of being contained by a scope, it is a child of a function.

The *Using* and *NamespaceAlias* classes are special as well because they have a position in the code, but neither is a real member in the usual sense. The *Using* class points to a namespace whose symbols can be used from the point of definition, or a single namespace member or a base class member that can be used with a modified accessibility (attribute *kind*), say. The *NamespaceAlias* class refers to a namespace and defines a new name (alias) for it.

The *Enumeration* class stands for C++ enums, and it stores an arbitrary number of *Enumerator*s shared with the parent scope (enums do not open a new scope). An *Enumerator* may have an explicit value that is represented by an expression connected to it by the *hasValue* composition. It also points to the *Enumeration* class it belongs to.

Our schema represents the two kinds of templates in C++ (class- and function templates) in a similar way by separating the template representation (the template parameters) from the actual template object as it does with the type representation. This template representation is called *ParameterList* and is located in the *templ* package (see Section 3.1.5). The two template classes are *ClassTempl* and *FunctionTempl*, which represent class templates and function templates, respectively.

Template specializations (classes *ClassTemplSpec* and *FunctionTemplSpec*) are handled in a similar way as templates. They are composed with the *ArgumentList* class from the *templ* package, which denotes the template specialization arguments. In addition, the template specialization points to the template it specializes.

Template instances can also be considered as members, although they do not have an exact position (they are not present in the source code directly, but are generated by the front end). Similar to the template specialization, the *TemplInstance* class is composed with the class *templ::ArgumentList*, and it points to the template or template specialization it is instantiated from.

Class *Asm* represents assembly declarations embedded in C++ code.

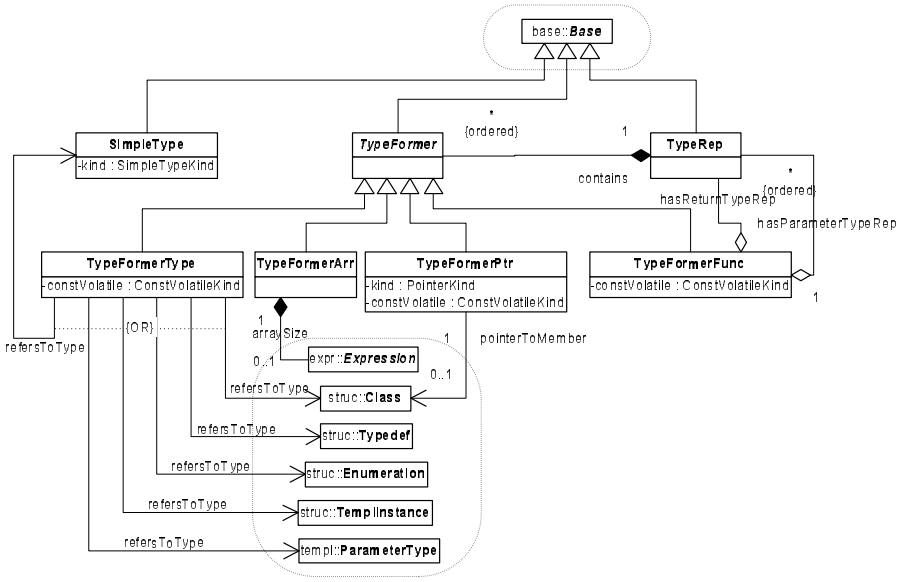


Figure 3.4: Class diagram of the ***type*** package.

3.1.4 *type* – the type package

The type representations (class *TypeRep*) are stored separately from the language entities which make use of them (e.g. functions and objects – see Section 3.1.3). This allows the possibility of storing each type representation only once, and to point to them from multiple nodes (see Figure 3.4).

The type representation class *TypeRep* is composed of small parts that we call *TypeFormer*s. These can be simple type formers like arrays (*TypeFormerArr*) and pointers/references (*TypeFormerPtr*); or type references (*TypeFormerType*), which can point to simple (built-in) types (class *SimpleType*) or some other entities that represent types like classes or typedefs. The *TypeFormerFunc* is a special type former, which is present when we are representing the types of functions or parts of a more complex type representation containing pointers to functions. It points to the type representations of the function's parameters and its return type. Optional const-volatile qualifiers that belong to a type are represented by an attribute in the *TypeFormerType* and *TypeFormerFunc* classes. The size of an array is captured by the composition *arraySize* from *TypeFormerArr* to *expr::Expression*.

The order of the type formers captures information on how the type is built up semantically. For example, the declaration `int *array[SIZE]` is represented by this *TypeRep*: (1) array of (2) pointers to (3) `int`s. Redundant information like parentheses are not captured.

Note that this type representation permits an arbitrary number of recursions for representing the types of parameters, say, that are pointers to functions which have parameters that are in turn pointers to functions.

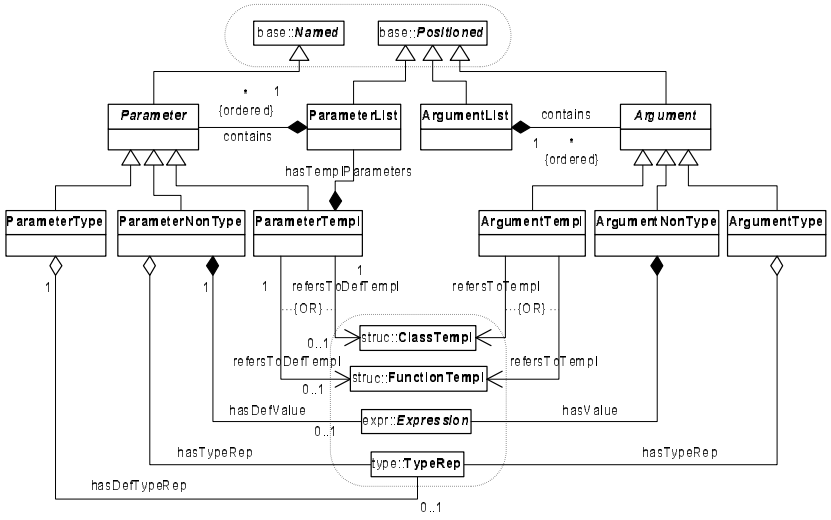


Figure 3.5: Class diagram of the ***templ*** package.

3.1.5 *templ* – the template package

There are two special language entities that class and function templates and template specializations possess: the template parameter list (represented by the class *ParameterList*) and the template argument list (represented by the class *ArgumentList*). These are shown in Figure 3.5 above. Both are composed with *Parameter*s and *Argument*s, respectively.

In C++ there are three kinds of template parameters: the "usual" type name, the non-type (i.e. value) and another template. The type name parameter is represented by the class *ParameterType*, which may point to a type representation as its default value. The non-type parameters are modelled by the class *ParameterNonType*, which represents its type in just the same way as the function parameters. It can have a default value, so it may contain a corresponding expression. Lastly, the parameter which is another template is represented by the class *ParameterTempl*, which simply stores its template parameter list in recursive form; it is composed with the *ParameterList* class. Additionally, it can have a default value, so it may point to the appropriate *struc::ClassTempl* or *struc::FunctionTempl* class.

Every template parameter has its corresponding template argument when the template is instantiated or specialized: the class *ArgumentType* represents a reference to a concrete type representation for the type name parameter. Similarly, the *ArgumentNonType* class contains a concrete expression and the *ArgumentTempl* class points to the concrete class or function template.

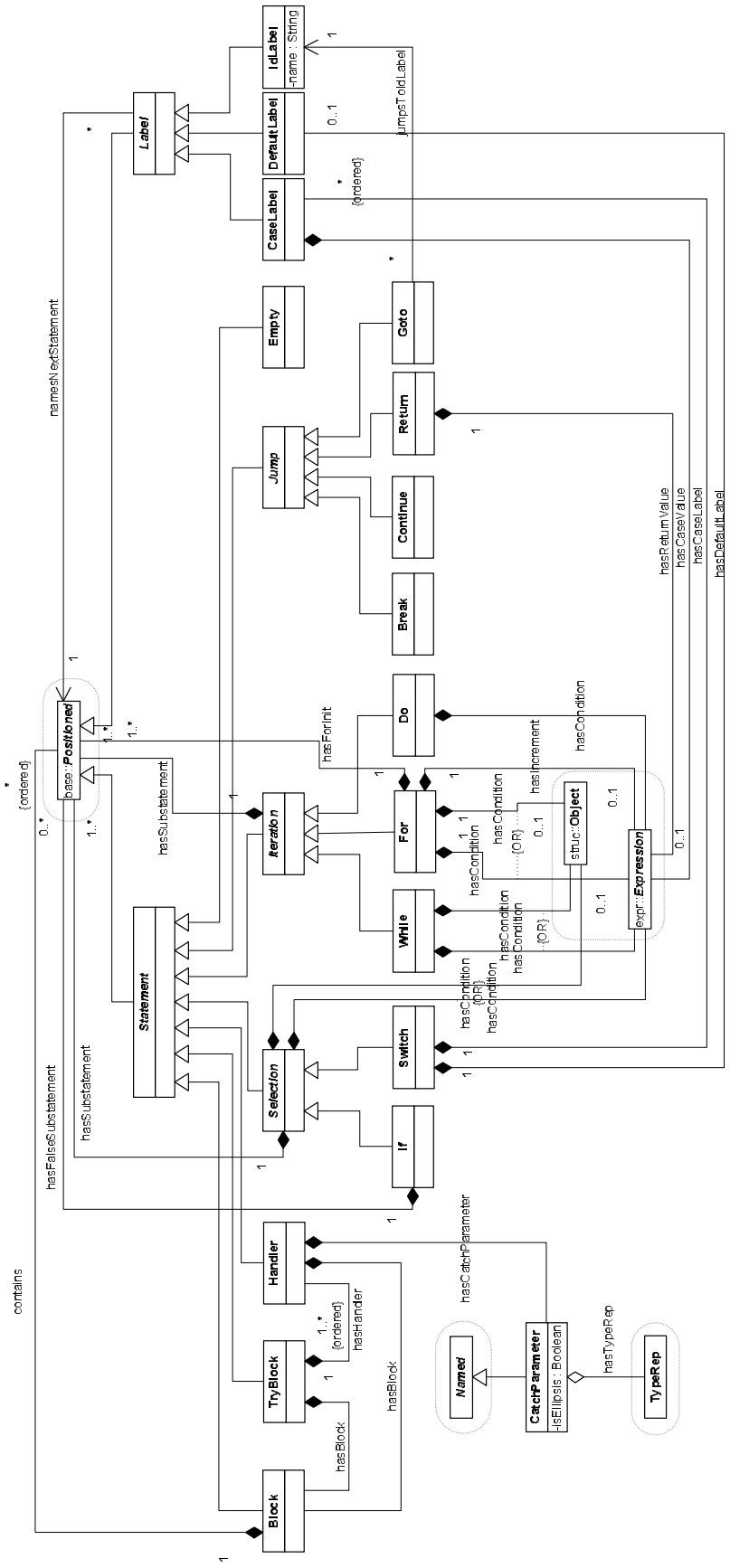


Figure 3.6: Class diagram of the ***statm*** package.

3.1.6 *statm* – the statement package

The classes in the *statm* package (see Figure 3.6) model the statements in C++ source code (instructions in a function body) and their scoping structure, which are organized around *Statement*, one of the child classes of *base::Positioned*. This abstract class is the parent of every kind of statement that may appear in a blockscope.

The first child of the *Statement* class is the class *Block*, which is a blockscope entity as well because it can be contained in another block. The composition from *Block* to *Positioned* enables the class to store other *Positioned* entities (like, for instance, *Statement*s, *expr::Expression*s and *statm::Object*s – local variables) in an ordered way. This recursive containment along with the other compositions in the package builds the skeleton structure tree of the modelled function body.

The *TryBlock* class is employed to represent try-blocks, so-called protected regions in C++ code whose purpose is to contain the "normal" program code without error handling instructions. The runtime errors are taken care of afterwards in one or more *Handler*s (also known as catch-blocks). The handlers always have a *CatchParameter* that carries the error itself (it is very similar to struc::Parameter). If the handler deals with every error then its *isEllipsis* property is true. The catch-parameter's type is represented (as usual in the schema) by *type::TypeRep*.

The C++ language offers the *Selection*, *Iteration* and *Jump* concepts for realizing the basic control flow programming language constructs. These are all represented by abstract classes having the same name in our schema.

The *Selection* class has a substatement, which may be a single instruction (statement, expression or local variable definition) or a block (containing further instructions and blocks); and a condition, which may be an expression (typically a logical one) or a *struc::Object* in the case when a local variable is defined as part of the condition.

There are two kinds of selections available in the C++ language. The first is the if-selection, which is represented by the *If* class that extends its base class with a false-substatement. The second one is the *Switch* selection which labels its substatements with *CaseLabel*s and an optional *DefaultLabel*. The *CaseLabel* class has a case value (typically an *expr:IntegerLiteral*).

The *Iteration* class represents every kind of loop and, in a similar way to *Selection*, it contains a substatement. The three concrete kinds of iterations are represented by the following three classes in our schema.

The *While* and *Do* classes are very similar to each other as they both extend their base class with a *condition*. In the case of *While* this condition may be also a struc::Object (local variable) definition. The *For* class represents the most complex iteration. It is composed of three parts: *initialization*, *condition* and *increment*. The initialization part typically contains an assignment-expression, and it may also contain local variable definitions (typically the loop variable). The condition part is the same as that for the *While* class. The increment part is an expression that typically increments the loop variable.

The *Jump* class models both the structured and unstructured jumps in the source code. The structured ones are the *Break*, *Continue* and *Return* statements. These are all very simple subclasses of *Jump*. *Return* may have a *return value*. The unstructured jump is represented by the *Goto* class, which points to class *IdLabel* that labels the statement where the control flow will be transferred.

The *IdLabel* class stores the label *name*. All three kinds of labels (*CaseLabel*, *DefaultLabel* and *IdLabel*) are subclasses of the abstract class *Label*, which points to the labelled statement.

Empty statements are modelled by the class *Empty*.

3.1.7 *expr* – the expression package

The classes in the *expr* package (see Figure 3.7) model the expressions in C++ source code and their syntactic structure (expression tree), which are organized around the class *Expression*, a subclass of *base::Positioned*. This abstract class is the parent of every kind of expression that usually appears in blockscopes but may also represent array sizes and initialization values, say. Every expression in our schema has a type representation (the type of the subexpression) modelled by the class *type::TypeRep*.

Class *Expression* has, among other things, three abstract subclasses which model three large groups of expression kinds: *Unary*, *Binary* and *Literal*.

The *Unary* class stands for operators in the C++ language that have one operand (which is connected by the composition called *contains*). Most of its subclasses are very simple, and their names describe their purpose (classes *FunctionCall*, *NamedConversion*, *CastConversion*, *PostIncDec*, *PreIncDec*, *TypeIdExpr*, *Indirection*, *AddressOf*, *UnaryArithmetic*, *UnaryLogical*, *SizeOfExpr* and *Delete*). Some of these classes have an attribute called *kind*, which refines their meaning (its type is one of the enumerations in the *base* package). The *FunctionCall* class is composed with a class called *ExpressionList* which stores the actual arguments of the function call. The function being called can be reached by traversing the *Id* expression contained in the *FunctionCall* class. The two conversion classes (*NamedConversion* and *CastConversion*) point to the corresponding *type::TypeRep* to which they convert the type of the operand. Class *Delete* has two properties. These say whether it is *global* and whether it deletes an *array*.

The *Binary* class models operators that have two operands (the composition called *contains* with multiplicity 2). Similar to the unary classes, most of its subclasses are simple, and their names describe their purpose (classes *Assignment*, *ArraySubscipt*, *MemberSelection*, *Comma*, *PointerToMember*, *BinaryArithmetic* and *BinaryLogical*). Some of these classes have an attribute called *kind*, which refines their meaning (e.g. whether the member selection is a dot or an arrow; compare this with the enumeration *base::MemberSelectionKind*).

The class *Literal* maps C++ literals to our schema. It has five subclasses (*IntegerLiteral*, *CharacterLiteral*, *FloatingLiteral*, *StringLiteral* and *BooleanLiteral*), which all

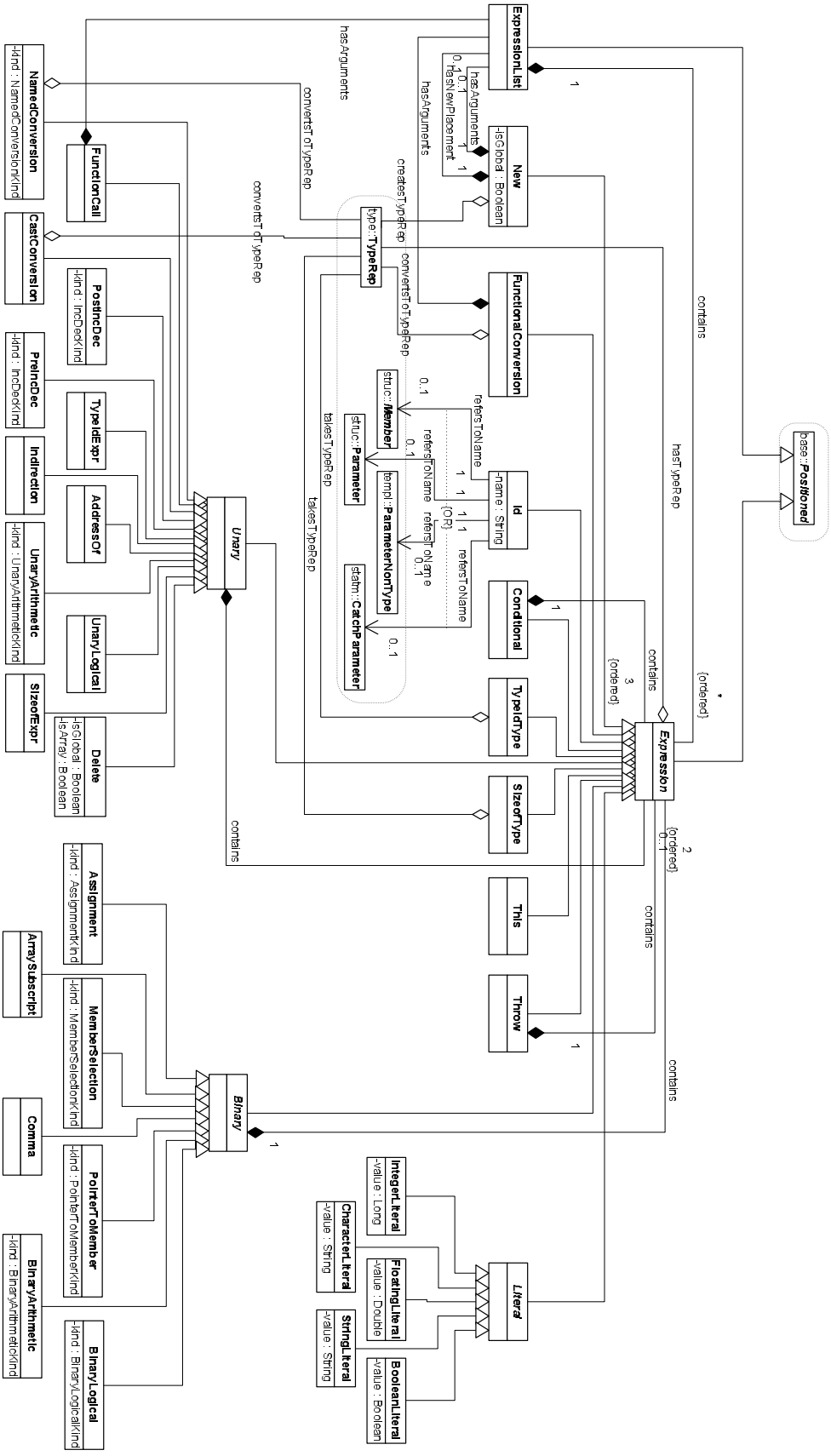


Figure 3.7: Class diagram of the *expr* package.

have an attribute with the same name (*value*) but with different basic types for storing the actual value.

The *ExpressionList* class is a composite that stands for an ordered list of expressions. This simplifies the modelling in several cases (e.g. representing the arguments of a function call).

Class *New* models the C++ `new`-operator. It has an attribute which stores information about whether it represents the global new-operator or not. It may have two optional *ExpressionList* objects to model the new-placement and the new-initializer arguments, respectively. It also points to the *type::TypeRep* class, which models the type of the object it creates.

The *FunctionalConversion* class stands for explicit type conversion (a simple type specifier followed by a parenthesized expression list). It points to the corresponding *type::TypeRep* to which it converts the type of the operand(s).

Class *Id* represents an identifier in an expression. Its *name* is stored as an attribute. It may point to a suitable program entity like struc::Member or struc::Parameter.

The class called *Conditional* models the three-operand selection expression "`?:`". It contains three sub-expressions represented as compositions.

The *TypeIdType* and *SizeOfType* classes are similar, so we will describe them together. Both are connected with the *type::TypeRep* class, which they take as an argument. *TypeIdType* and *SizeOfType* model the C++ operators `typeid` and `sizeof`, respectively, in the case when the operand is a type. (In the case when the operand is an expression these are modelled by the classes *TypeIdExpr* and *SizeOfExpr* described above.)

Class *Throw* represents the `throw`-operator, which may contain an *expr::Expression* modelling the expression being thrown.

The class called *This* stands for the C++ `this` primary expression.

3.1.8 Example schema instance

We illustrate the use of our schema through an example instance of it. We will use the example C++ source code given in Figure 3.8. The ASG for the example is given in Figure 3.9. We use an object diagram-like notation, where the object instances of the schema's classes are represented and the links that connect them clearly show the instances of various association and aggregation relations. The ordered associations are represented by numbering the links. We have simplified the diagram for clarity by omitting attributes such as line numbers, which are not necessary for comprehension.

The schema uses integers as unique identifiers/keys for nodes; e.g the key of the topmost node is "1." The class of each node is given to the right of the key number; for instance, the class of node 1 is *struc::Namespace*. The schema uses a *name* attribute in some nodes to give the name of the source item being represented. For example, the name attribute of node 1 is *"global namespace."*

```
template <typename T, int Size>
class Array {
  T arr[Size];
public:
  virtual const T& get(int idx) const {
    return arr[idx];
  };
  virtual void set(int idx, const T& val) {
    arr[idx] = val;
  }
};
```

This example implements a generic array which expects two parameters (the stored element type and the array size) and has two public methods called ***get*** *and* ***set****.*

Figure 3.8: C++ source code example.

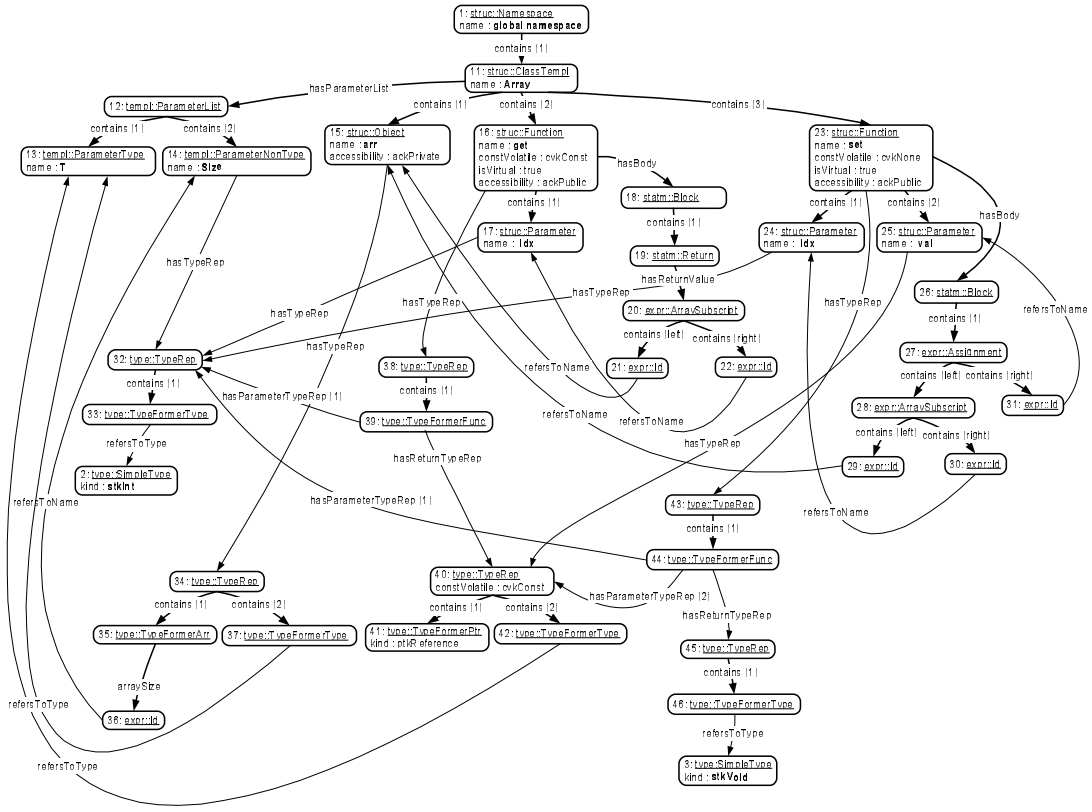


Figure 3.9: Schema instance for the example.

The template class *Array* is represented by node 11. It contains a template parameter list (node 12), which has two children that represent the two template parameters. The first one is the typename *T*, while the second one is the non-type parameter (constant value) *Size*, which is – similar to simple variables – typed. Its type is represented by *TypeRep* node 32 (we will return to type representations later). As the reader will notice, the template representation is completely separate from the basic scoping structure of the ASG. So it can easily be replaced by another kind of template description if required.

The template class *Array* has three children (ordered): object *arr* (node 15) and functions *get* (node 16) and *set* (node 23). The type of object *arr* is represented by *TypeRep* node 34. The function called *get* is virtual and constant. These facts are stored as attributes in node 16. The function has one parameter with the name *idx* (node 17). The type of this node is the same as that for the template parameter *Size*, so it points to the same *TypeRep* 32. In this way there will be only one instance of each unique type representation and it is straightforward to compare the types of any two different C++ entities. We will now elaborate on the type representation of the *get* function (node 38).

TypeRep 38 contains a *TypeFormerFunc* node 39, indicating that it is the type representation of a function. It points to the *TypeRep* of the function's return type (node 40) and the *TypeRep* of the parameter *idx* (node 32). This is needed because the type of a function includes the whole signature (return type and parameter types). The return type representation (node 40) stores the 'constant' property of the return type and contains two type formers for the type itself. The first one is a pointer (reference) former (*TypeFormerPtr* node 41), meaning that the return type is a reference to a type. The second type former is a *TypeFormerType* (node 42), which points to template parameter *T* (node 13). So the overall meaning of the type representation is a reference to template parameter *T*. The type representation of the parameter of function *get* is modelled by node 32. This is a very simple *TypeRep* containing only one *TypeFormerType* that points to the *SimpleType* representing the built-in type *int*. As can be seen, like that for the template representation, the type representation is also completely separate from the rest of the ASG so it can easily be modified or replaced by another approach for describing types if needs be.

The body of function *get* is represented by node 18, which is a block statement (*statm::Block*). This block is rather simple, containing just one return statement (*statm::Return* node 19). The return value is represented by node 20, which is an array subscript expression (*expr::ArraySubscript*) whose left and right operands are of the kind *expr::Id* that simply point to nodes *arr* (15) and *idx* (17), respectively.

The representation of the function called *set* is similar to that described earlier, so we will not elaborate on it here.

3.2 Data exchange with other tools

Successful interchange of the data created by Columbus according to the Columbus Schema for C++ was achieved in several studies.

The first application was created in cooperation with the Nokia Research Center for Nokia's proprietary UML design environment *TDE* [65]. We used the Columbus framework as the C++ analyzer front end of TDE. The built-up schema instances were converted to UML class diagrams and transferred to TDE though a COM interface.

The *Maisa* [52; 55] project of the University of Helsinki for recognizing standard Design Patterns [34] in C++ programs also successfully utilized the output created by Columbus (see Chapter 5).

Another example of the schema's use was in a FAMOOS project with the Crocodile metrics tool [58]. An important application is the currently ongoing work on the exchange of data between Columbus and the GUPRO tool [20], which uses GXL as its input format. We also achieved a successful interchange with the *rigi* graph visualizer tool [51; 57] (see Section 4.2.6). Recently, we started a joint study with the University of Waterloo in Canada to visualize the Columbus schema instances in PBS, the Portable Bookshelf [31].

3.3 Summary

This work was motivated by the observation that successful data exchange is crucial for re- and reverse engineering tools. This requires a common format, one that can be used in various tools like front ends and metrics tools. A standard schema still has to be found. In this work we propose an exchange schema for the C++ language called the Columbus Schema for C++ for this purpose.

The schema is modular, thus providing additional flexibility for its extension or modification. It captures the (preprocessed) C++ language entities in fine-grained detail and also contains higher-level elements. The description of the schema is given using UML class diagrams, which facilitates its simple implementation and easy physical representation (e.g. using GXL).

The schema was implemented in the reverse engineering framework Columbus and we also provide several data exchange examples with other tools.

"The science of today is the technology of tomorrow."
Edward Teller

Chapter 4

Fact extraction and presentation

Fact extraction is, in our understanding, a process which prescribes certain steps that describe the way information about source code can be obtained. These steps include acquiring project/configuration information, the analysis of the source files with analyzer tools, the creation of some kind of representation of the extracted information, the merging of these representations and various conversions made on this merged representation to enable the actual use of the extracted information.

The goal of our extraction process is to build up the appropriate schema instances and fill them with information about the analyzed software system so that various analyses and calculations can be performed on them.

The most difficult steps in the process are acquiring project/configuration information and a proper analysis of the source files. In this chapter we will describe methods with which these tasks can be done with relative ease.

We will also describe the whole extraction process and the tools we developed to support it.

4.1 The fact extraction process

An outline of the process of fact extraction and presentation can be seen in Figure 4.1. The process consists of five consecutive steps where each step uses the results of the previous one [29]. In the following, these steps will be specified in detail.

An important advantage of this approach is that the steps of the process can be performed *incrementally*, that is, if the partial results of the certain steps are available and the input of the step has not been altered, these results do not have to be regenerated. This is particularly important in the case of extracting facts from large, real-world software systems.

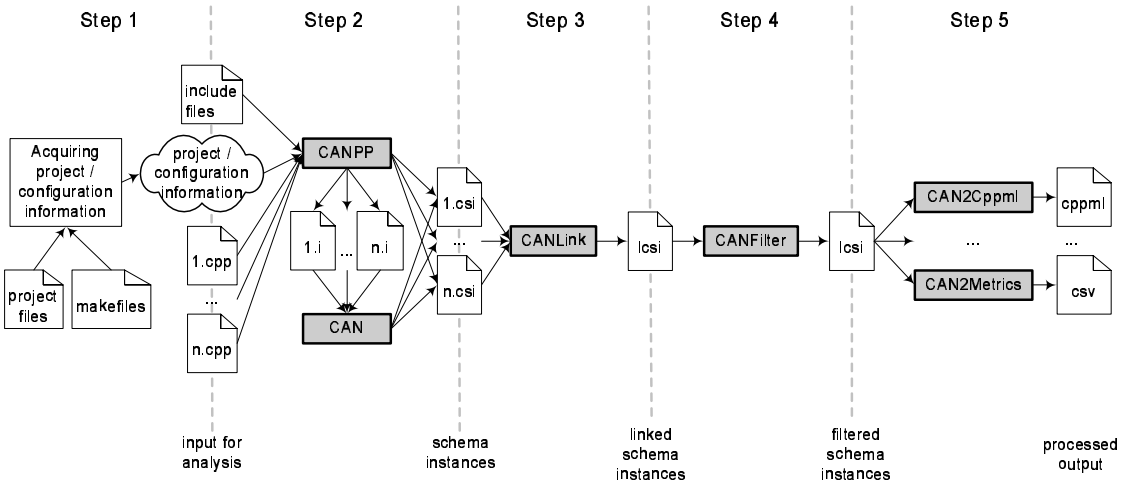


Figure 4.1: The fact extraction process.

Step 1: Acquiring project/configuration information

As already discussed in Section 2.5, acquiring project/configuration information is indispensable in carrying out the extraction process. The source code of a software system is usually split into several files and these files are arranged into folders and subfolders. In addition, different preprocessing configurations can apply to them. The information on how these files are related to each other and what settings apply to them are usually stored in *makefile*s (when building software with the *make* tool) or in different *project files* (when using different *IDEs – Integrated Development Environments*). Acquiring this information from these files is a non-trivial task as different IDEs use different (and in most cases undocumented) file formats. Makefiles present additional difficulties: getting information out of them is extremely hard because they are not just suitable for code compilation; they can perform any arbitrary task.

We introduce a so-called *compiler wrapping* method for using makefile information (see Section 4.2.3) and two different approaches for handling IDE project files: *IDE integration* (see Section 4.2.2) and *project file import* (see Section 4.2.1).

Step 2: Analysis of the source codes – creation of schema instances

In this step the input files are processed one by one using the project/configuration information acquired in the first step (for instance, macro definitions to be used and paths to different header files to be included). First, the preprocessing and extraction of preprocessing-related information are carried out (with the *CANPP* tool in our case – see Section 4.2.4). Second, the preprocessed file is handed over to the C++ analyzer (to the *CAN* tool in our case – see Section 4.2.4), which then analyzes the file and extracts C++ language-related information from it. Both tools create the corresponding schema instances and save them to appropriate files. This step is performed for every single input file.

Step 3: Linking of schema instances

After all the schema instance files have been created the linking (merging) of the related schema instances is done (with the *CANLink* tool in our case – see Section 4.2.4). This way, similar to real compiler systems that create different files which contain C++ entities that logically belong together (like libraries and executables), the related entities are grouped together. The outputs of this step are the merged schema instances for each logical unit (subsystem) of the analyzed software system. These merged instances can, of course, be further merged into one single schema instance to present the software system as a whole.

Step 4: Filtering the schema instances

In the case of really large projects the previous steps can produce large schema instances which contain huge amounts of extracted data. This is difficult to present in a useful way to the user (he/she is usually interested only in parts of the whole system at any given time). Different methods can help in solving this problem, like selecting only a particular module for further processing (see Section 4.2.5 for our solution).

Step 5: Processing the schema instances

Because different C++ re- and reverse engineering tools use different schemas for representing their data, the (filtered) schema instances must be converted to different formats to be widely usable (we can convert the extracted facts into various formats and also apply different computations on them: see Chapters 4.2.6 and 4.2.7 for details).

4.2 The Columbus framework

The fact extraction process introduced in the previous section is well supported by our reverse engineering framework, which we present here.

We developed the *Columbus Reverse Engineering Framework* [22–26; 28; 29] in an R&D project with the Nokia Research Center. The main motivation behind developing the framework was to create a toolset which supports fact extraction and provides a common interface for other reverse engineering tasks as well. The main tool is called Columbus REE (Reverse Engineering Environment), which is the graphical user interface shell of the framework (see Figure 4.2). Columbus REE is not limited to the C++ language; all C++ specific tasks are performed by using different plug-in modules of it. Some of these plug-in modules are present as basic parts, but the REE can be extended to support other languages and reverse engineering tasks as well. Doing this, we have obtained a versatile and easily extendible environment and framework for reverse engineering. The actual analysis and fact processing is done by different command line tools invoked by Columbus REE (these tools run on both Windows and GNU/Linux operating systems).

The Columbus framework currently includes the following tools, which are listed and discussed in the following:

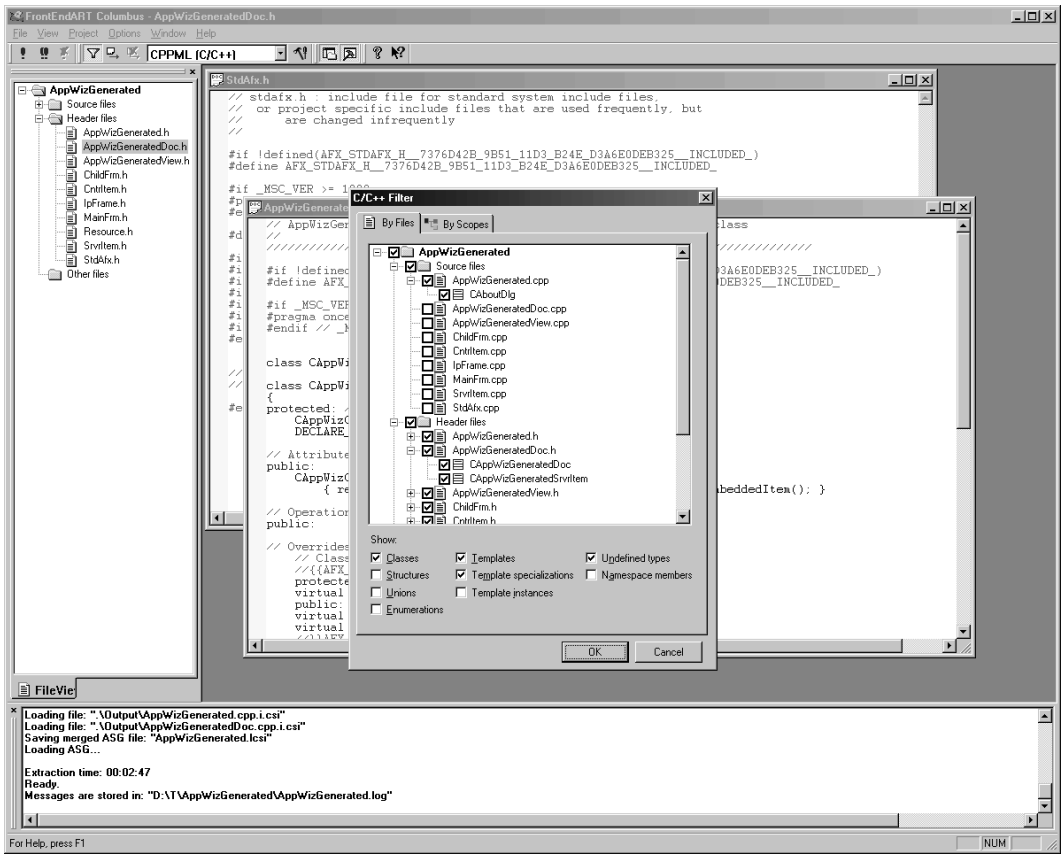


Figure 4.2: The Columbus Reverse Engineering Environment

- *Columbus REE*. The graphical user interface shell of the framework (see Section 4.2.1).
- *Columbus IDE Add-ins*. Graphical user interface shell functionality of the framework in different IDEs (see Section 4.2.2).
- *CANGccWrapper toolset*. GCC compiler-wrapper tool with different helper scripts (see Section 4.2.3).
- *CANPP*. C/C++ preprocessor and preprocessing schema instance builder tool (see Section 4.2.4).
- *CAN*. C++ analyzer and schema instance builder tool (see Section 4.2.4).
- *CANLink*. C++ schema instance linker tool (see Section 4.2.4).
- *CANFilter*. C++ schema instance filter tool (see Section 4.2.5).
- *CAN2**. C++ schema instance converter and processor tools (see Sections 4.2.6 and 4.2.7).

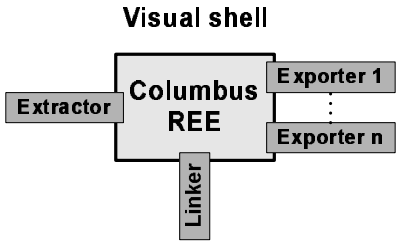


Figure 4.3: The general structure of Columbus REE.

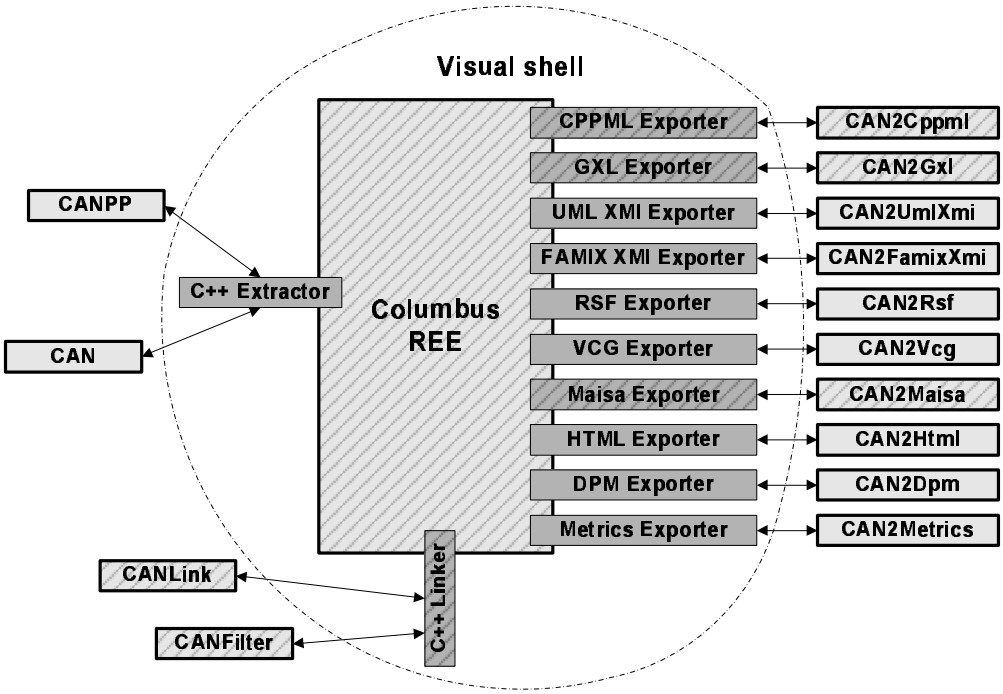


Figure 4.4: The C++ specific configuration of Columbus REE.

4.2.1 Columbus REE (Reverse Engineering Environment)

The Columbus REE (see Figure 4.2) is a general reverse engineering environment which runs on Microsoft Windows. All C++ specific tasks are performed by its plug-in modules (DLLs – dynamic link libraries).

The operation of the Columbus REE is performed via three types of plug-in modules. The reader should look at Figure 4.3 for the general concept and Figure 4.4 for the actual C++ specific configuration. The author implemented the parts of the environment which are filled with striped lines. The three types of plug-in modules are the following:

- *Extractor plug-in modules* – The task of an extractor plug-in module in general is to properly analyze a given input source file and to create a file which contains the extracted information. More precisely, the extractor module for C++ invokes the

CANPP and *CAN* tools to preprocess and analyze the input file, and to create the files containing the appropriate schema instances. This plug-in carries out Step 2 of the extraction process.

- *Linker plug-in modules* – The task of a linker plug-in module in general is to link the schema instance files created by the extractor plug-in module into a larger instance which represents the whole analyzed software system (or a part of it). This plug-in module may also carry out the filtering of the linked schema instance in order to produce a more clear-cut representation for exporting. More precisely, the linker module for C++ invokes the *CANLink* tool to link the files created by *CANPP* and *CAN*. It afterwards invokes the *CANFilter* tool as well, which displays a visual filter with which it is possible to filter the schema instance. This plug-in takes care of Steps 3 and 4 of the extraction process.

- *Exporter plug-in modules* – The task of an exporter plug-in module in general is to convert the schema instance (built up and filtered by the linker plug-in module) to some other format. The exporter modules for C++ provide converters for four XML formats (CPPML, GXL, UML XMI and FAMIX XMI), RSF, VCG, Maisa and HTML. It is also possible to do arbitrary processing on the schema instances. For instance, the framework contains plug-ins for calculating object-oriented metrics, recognizing object-oriented design patterns and code auditing. This plug-in carries out Step 5 of the extraction process.

Besides the basic plug-in modules the user can easily write and add his/her own new plug-in module to the Columbus REE. By packaging every C++ functionality into command line tools we made most of the tools platform-independent.

To help acquire project/configuration information (Step 1 of the extraction process) the Columbus REE can import Microsoft Visual Studio 6.0 and .NET project files. For the case where there is no project information available at the start, a so-called *Project Setup Wizard* is available, with the help of which a project can easily be set up from scratch.

4.2.2 Columbus IDE Add-ins

A significant part of Columbus REE is engaged in managing the project (configuration) of the files to be analyzed. This work is also done by the popular IDEs, so it was logical to package the remaining part of the Columbus REE – the part that deals with the extraction process – into a separate component called *Columbus DLL*, which communicates with different so-called *Columbus Add-ins* for IDEs. These add-ins are basically plug-ins that extend the functionality of the IDEs (similar to the Columbus REE plug-ins). Please look at Figure 4.5 for the structure of the add-in system. The author implemented the parts which are filled with striped lines.

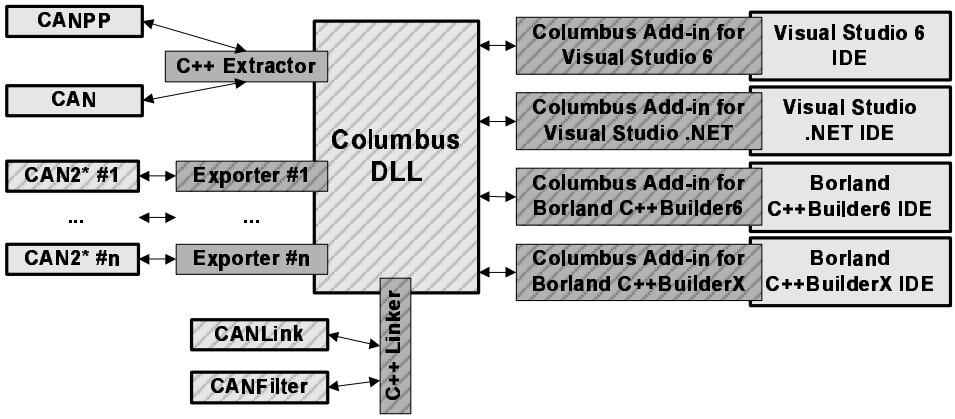


Figure 4.5: Columbus Add-ins for IDEs.

In this case our tool appears as a new toolbar within the IDE and its operation is very similar to the usual build process. The active project under development is analyzed with a simple click of a button and the output can be converted to any format supported by the Columbus framework. Currently Microsoft Visual Studio 6.0/.NET and Borland C++Builder 6/X are supported.

4.2.3 Compiler wrapping

As we have already mentioned in Sections 2.5 and 4.1 when discussing the topic "Acquiring project/configuration information", the source code of a software system is usually split into several files and these files are arranged into folders and subfolders. Furthermore, different preprocessing configurations may apply to the source. In this section we deal with the case where the information on how these files are related to each other (and what settings apply to them) is stored in *makefile*s (used by the *make* tool). An important aim that we set ourselves was to *not change* anything in the subject system (not even the makefiles). The method described in the following successfully fulfills this requirement. It was tested with the GCC compiler in the GNU/Linux environment, but the idea is also applicable to other compilers and operating systems.

The make tool and the makefiles represent a powerful pair for configuring and building software systems. Makefiles may contain not only references to files to be compiled and their settings, but also various commands like those invoking external tools. A typical example is when the source file to be compiled is generated on-the-fly from IDL descriptions by another tool. These possibilities are a headache for reverse engineers because every action in the makefile must somehow be simulated in the reverse engineering tool. This may be extremely hard or even impossible to do in some cases.

We approached this problem from the other end and solved it by "wrapping" the compiler. This means that we temporarily hide the original compiler by using a wrapper toolset. The structure of the compiler wrapper toolset is shown in Figure 4.6. The

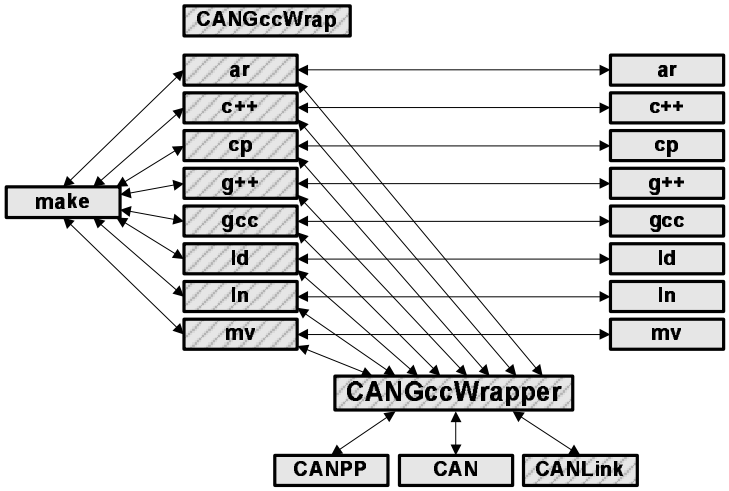


Figure 4.6: GCC compiler-wrapper toolset.

author implemented the parts which are filled with striped lines. This toolset consists of several scripts and a program called *CANGccWrapper*. Among the scripts there is a key one called *CANGccWrap*, which is responsible for hiding the original compiler by changing the PATH environment variable. Actually, all the user has to do is to run this script. The script inserts the path of the folder in which the other scripts can be found to the beginning of the PATH environment variable. The names of the other scripts correspond to the executable files of the compiler (for instance *gcc*, *ar* and *ld*). If we want to execute a program, the operating system searches for it in the folders given in the PATH variable in the same order. This means that if the original compiler should be invoked, our wrapper script will start instead of it because it appears first in the PATH variable. If we do not want to use the wrapper anymore it can be simply switched off.

The other scripts are quite similar to each other, the only difference being that they "hide" different tools of the compiler. The scripts first execute the original compiler tool (for instance *g++*) with the same parameters and in the same environment so the output remains the same, hence we do not notice that not the real compiler was called first. The scripts also examine whether the original program terminated normally or not and they terminate with the same value. This is very important because complex systems usually run various tests before compilation to determine the compiler capabilities and environment settings. They usually do this by trying to compile small programs containing the tests and examine the termination of the compiler. If the scripts do not take into account the results of the compiler and always terminate normally even when the compilation failed, say, the compilation will be misled and this will probably cause problems later on during compilation.

After calling the original compiler, the scripts also call the program *CANGccWrapper*, which in turn calls the corresponding analyzer tool (*CANPP*, *CAN* or *CANLink*). Since the command line switches of the analyzer tools are not the same as the compiler's,

they cannot be easily called directly from the scripts like the compiler. Another problem is that we use different tools for different tasks (for instance, *CANPP* can be used only for preprocessing-related issues), while *gcc* can be used for several purposes like preprocessing, compiling and linking, depending on with which switches and parameters it is invoked. So we must examine the parameters to choose the tool(s) which must be called (for instance, "gcc -E ..." means "do not compile the file just preprocess it"). The *CANGccWrapper* program solves these problems. The scripts call *CANGccWrapper* with the same parameters as the compiler and *CANGccWrapper* will call the appropriate analyzer tools.

We successfully used this approach with GCC on the GNU/Linux platform for extracting information from the open source real-world software system called *Mozilla*. This proves the operability of this method (see Chapter 6).

4.2.4 Analyzer tools

Source code analysis in the traditional sense is performed by three command line tools that are described in the following.

CANPP (C/C++ ANalyzer-PreProcessor) is a command line program for analyzing C/C++ preprocessor-related language constructs and for preprocessing the code. The input is a C/C++ source file with various settings (like include paths and macro definitions), and the outputs are the preprocessed file and the built-up instance of the Columbus Schema for C/C++ Preprocessing of the source file. The preprocessor is *fault-tolerant*, that is it has the ability to parse incomplete, syntactically incorrect source code as well.

CAN (C++ ANalyzer) is a command line program for analyzing C++ code. CAN processes one compilation unit at a time (a preprocessed source file), and the output is the built-up schema instance of the analyzed compilation unit. The information collected by CAN corresponds to the Columbus Schema for C++ (see Section 3.1). The C++ language processed by the analyzer meets the ISO/IEC standard of 1998 [41]. Moreover, this grammar has been extended with the Microsoft extensions used in Visual C++, the Borland extensions used in C++Builder and the GCC extensions used in g++. CAN supports the *precompiled headers* technique too, which is used by some compiler systems as well in order to decrease compilation time. This technique is especially useful for large projects. The parser is *fault-tolerant* (it can parse incomplete, syntactically incorrect source code), which means that it can carry on with the analysis from the next parsable statement after the error.

CANLink (CAN Linker) is a C++ schema instance linker tool. Similar to compiler linkers, it merges the instances of the Columbus schemas into larger instances. This way, C++ entities that logically belong together like libraries and executables are grouped into one instance. These merged instances can, of course, be further merged into one single schema instance to represent the software system as a whole.

4.2.5 Filtering

The fact extraction process can produce huge amounts of extracted data, especially in the case of large subject systems, which is hard to present in a way that can offer useful information to the user (the user may be interested only in parts of the system at any given time). Different filtering methods in the *CANFilter* tool can help in solving this problem.

There are three options for filtering:

- *Filtering using C++ entity categories*, like classes, templates and enumerations. With this option all elements that do not belong to the selected categories will be filtered out.

- *Filtering by input source files*. All C++ elements that come from the input files which are not selected are filtered out. This way all elements that come from system libraries, say, can be easily filtered out (these header files are not strictly part of the analyzed software system).

- *Filtering according to scopes*. Different C++ elements like classes and namespaces can be selected/deselected individually in a tree-view browser that represents the scoping structure of the project.

4.2.6 Schema instance conversions

Because different C++ re- and reverse engineering tools use different schemas for representing their data, the schema instances built from the extracted facts by Columbus can be converted to other formats to achieve tool interoperability.

We can convert our schema instances to several formats. Some of these are general, while others are tool specific. In the following we will briefly describe these formats.

CPPML

This conversion permits the creation of XML documents in the form of *CPPML* – C++ Markup Language. Their structure is based on the Columbus Schema for C++. Appendix A contains the DTD – Document Type Definition of the CPPML language.

GXL

With this conversion GXL representations can be created from the information that has been extracted. *GXL* – Graph eXchange Language [39] is an XML-based graph description format. Since the Columbus schemas basically define graphs, this format is ideal for representing them in a convenient way.

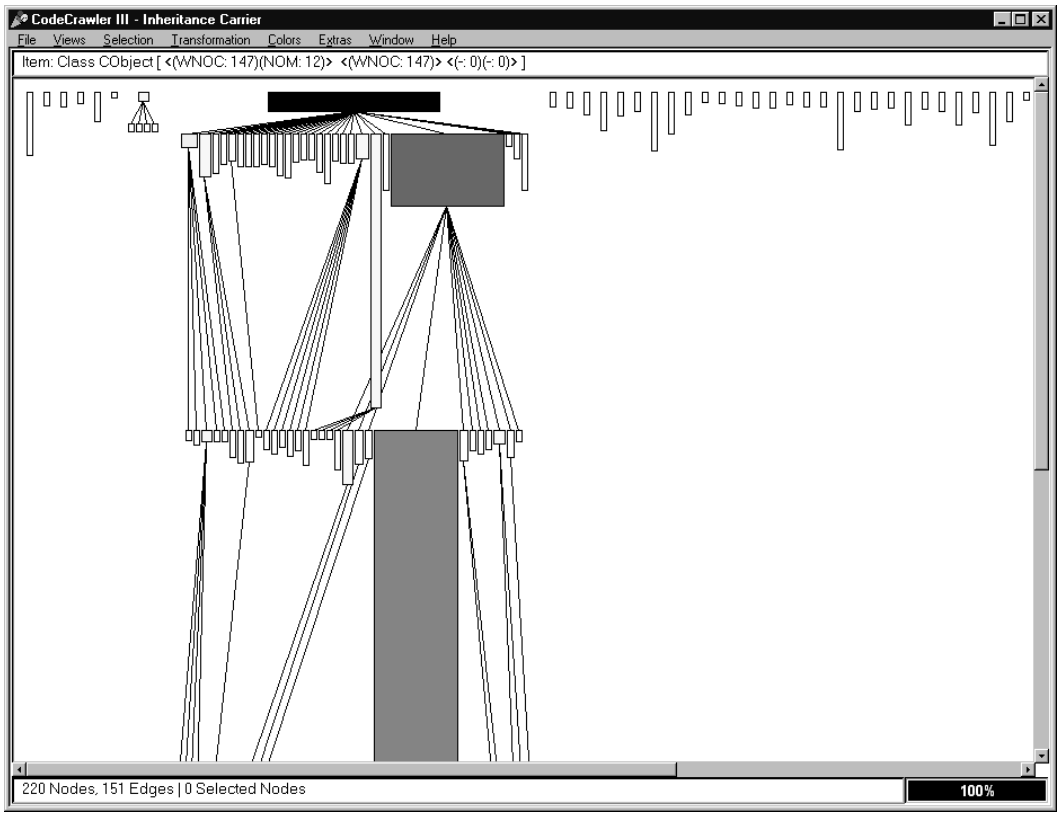


Figure 4.7: Class inheritance visualization in CodeCrawler

UML XMI

This conversion calculates UML class diagrams from the Columbus schema instances. The diagrams are represented as standard *UML XMI* [54] documents (versions 1.0 and 1.1 are supported). The XMI documents can be further processed with any XMI enabled tool (like Rational Rose or Together ControlCenter).

FAMIX XMI

With this conversion a *FAMIX* XMI [19] representation of the extracted information can be created whose information content is similar to that of the UML XMI representation (see above). This format can be utilized with the *CodeCrawler* tool [19] for visualization and metric calculations (see Figure 4.7, where an example class inheritance visualization is shown combined with class sizes).

RSF

The Columbus Framework also has converters for creating *RSF – Rigi Standard Format* documents for the *rigi* tool [51] with three different contents: (1) a graph based on the

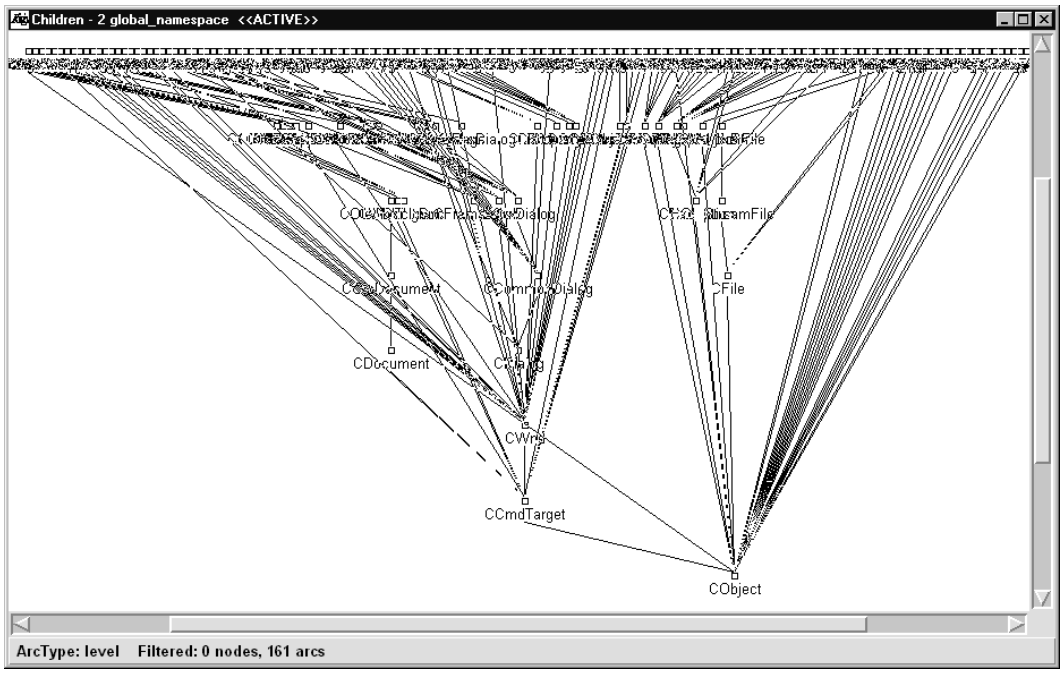


Figure 4.8: Class inheritance visualization in rigi

Columbus Schema for C++, (2) a call graph and (3) a UML class diagram-like graph. All of these use different rigi domains which can be created with the Columbus REE as well. Figure 4.8 shows an example class inheritance visualization in rigi.

VCG

With this conversion, VCG graph representations can be created. *VCG* – Visualization of Compiler Graphs [59] is a text-based graph description format. Like GXL, this format is also suitable for representing the schema instances in a handy way.

Maisa

This conversion creates a PROLOG-like document for the metrics and design pattern miner tool *Maisa* [27]. This makes it possible for us to recognize design patterns in C++ code with the integration of the Columbus framework and Maisa (see Chapter 5 for details).

HTML

This conversion can be used to create a hypertext documentation of the extracted project in *HTML* form. The generated documentation presents the project in a user-friendly fashion, all the necessary information about the classes and other entities being shown in a structured form. Three types of browser frames are also supplied with which

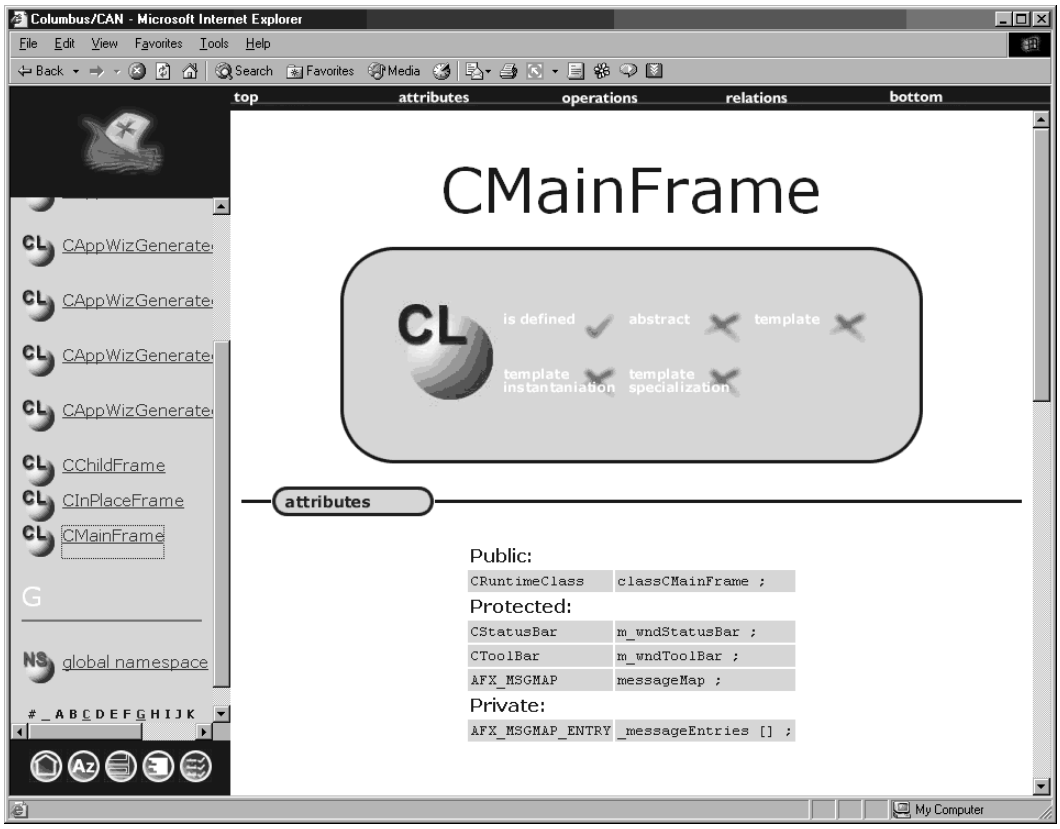


Figure 4.9: A class documentation in a browser

a project can be easily navigated. These display the classes using (1) their names in alphabetical order, (2) their scoping structure and (3) their inheritance relationships. Figure 4.9 shows a class documentation displayed in a web browser.

4.2.7 Derived outputs

We also use our schema instances to prepare different derived outputs. This means applying further computations on the instances, such as calculating metrics and recognizing design patterns.

The schema instances holding the facts extracted from the subject system can be processed either directly in memory using the framework's API (see Section 3.1), or using the CPPML/GXL/RSF/VCG outputs, which are identical mappings of the schema instances to files (see Section 4.2.6).

The following derived outputs are directly calculated from the schema instances and can be employed, for instance, in applications whose aim is to measure or improve software source code quality.

Metrics

With this processing 88 different object-oriented metrics can be calculated (system level, class level and function level metrics). These are saved to standard CSV (Comma Separated Value) files, which may be further processed using spreadsheet editors, statistical programs or something similar.

Design patterns

This processing looks for design pattern [34] instances in C++ source code. It introduces a new approach to the problem of pattern detection because it includes, among others, the detection of call delegations, object creations and operation redefinitions. These elements identify pattern occurrences more precisely. The pattern descriptions are stored in an XML-based format, the *Design Pattern Markup Language* (DPML). This gives the user the freedom to modify the patterns, adapt them to his or her own needs, or create new pattern descriptions [3] (see Chapter 5 for details).

SourceAudit

This processing does code auditing that can investigate the schema instances (and indirectly examine the source code as well) and check them against a set of rules that describe the preferred properties of the code. These rules deal with issues related to coding style and dangerous code constructs, so they effectively extend the warning reporting capabilities of the compiler. This way it can help in increasing the quality and reliability of the code. A special-purpose tool called *SourceAudit* [24; 25] was developed on top of the Columbus fact extraction-, IDE integration- and code auditing technology. Code auditing rules are organized into rule packages, of which there is a built-in one called *general*. Further rule packages can be plugged in separately, so the program can be easily extended at any time with new packages.

4.3 Summary

In this chapter we presented our approach that lists five steps which have to be done to successfully carry out an automated fact extraction task. The chapter also discussed the Columbus reverse engineering framework in detail, which supports the process. The framework contains various tools and extension mechanisms so it relieves researchers of the burden of having to write parsers for different purposes and allows them to concentrate on their own concrete research topic. The framework is widely used at universities across the world, and so far several hundred downloads of the framework have been registered.

The main advantage of this work is that it offers a process and an extendible framework for fact extraction, representation and conversion. The framework already supports several popular tools like *rigi* and *CodeCrawler*, but it could be easily extended to support any other re- and reverse engineering tool as well.

Part II

Utilization of the extracted facts

"Consider the past and you shall know the future."
Chinese proverb

Chapter 5

Recognizing design patterns in C++ source code

Due to the increase in size and complexity of software systems, the importance of being able to comprehend and assess the quality of (legacy) software source code has been steadily rising.

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 facts from the source code and represent them as a set of UML diagrams [53]. 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 current object-oriented software engineering, *design patterns* [34], is almost totally without advanced tool support. By abstracting practical solutions for frequently occurring design problems to an object-oriented format, design patterns are the most natural and useful assets when recovering the architectural design and the underlying design decisions from the software code.

Design patterns are micro architectures that have proved to be reliable, easy-to implement and robust. Hence they can be a measure of the quality of an object-oriented software system. So a software system can be characterized, among other ways, by the number of design patterns used. Of course, one must fully understand the design patterns one would like to use because, if improperly used, they might result in unnecessarily huge class structures that could in the worst case even decrease code quality.

The recognition of design patterns is a crucial question in reverse engineering as they represent a high level of abstraction in object-oriented design. As mentioned above, one possible usage might be to measure the quality of a software system. This could help the project managers to decide whether a piece of code is good enough for use in a project. Here, "good enough" means that the code should be readily understandable, and it should be easy to modify parts of the code without needing to modify the whole code (i.e. the program should have a modular design). If the design patterns are well

documented, it should be much easier to understand the source and to make appropriate modifications on a well-defined part of it.

Another possible usage is to help document a source without proper comments on patterns to gain the advantages described above. Well-commented program code is much easier to maintain than code without comments or with poor comments. We can find pattern instances and, this way, help insert comments where necessary. Yet another possible usage is in forward engineering, where the system designers inspect the source to see if the coders have implemented the pattern correctly.

Design patterns are described by listing the intent, motivation, applicability, structure (with UML diagrams), participants, collaborations, consequences, implementation details, sample code, known uses and related patterns. All of these except the sample code are written for humans, but they do not prescribe how the pattern will be implemented. Even sample code is only useful for comparing structures, as function names and implementations may differ. So we must find a way to efficiently describe the patterns and then find a way to compare these pattern descriptions with the code. It is obvious that a direct comparison with the pure code will not work. The problem is to not only find an intermediate format in which patterns can be described and to which the code can be relatively easily converted, but also to construct an algorithm that can efficiently find patterns in the transformed code.

There are three kinds of design patterns:

- *Creational* – these patterns are concerned with the creation of objects. They may determine the type of the object and its multiplicity. Here it is not enough to match the pattern structure because the real functionality is hidden in the function implementations. Though it is difficult to recognize these patterns it is fortunately not impossible because object creations can be identified in the source code.

- *Structural* – these patterns deal with the composition of classes or objects. They define class hierarchies and various relations. In these patterns most features are described with the declarations of the operations and attributes, so they are easier to recognize than the creational ones.

- *Behavioral* – these patterns describe how classes interact and how their tasks are to be assigned. Hence the behavior of the patterns are defined in the bodies of the operations, the knowledge of the declarations is insufficient. This makes these patterns the most difficult to recognize.

In this chapter we present two methods for automatically recognizing design patterns from object-oriented (C++) code. These are the following.

- *Integration of Columbus and Maisa* [27]. The method relies on two software tools, *Columbus* (see Section 4.2) and *Maisa* [52; 55]. 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, the Columbus-Maisa pair is able to recognize design patterns in C++ code (see Section 5.2).

- *Pattern miner algorithm in Columbus* [3]. We designed and implemented an algorithm within the Columbus framework, which presents a new approach to the problem of design pattern recognition. The algorithm uses more precise pattern descriptions than previous approaches because call delegations, object creations and operation redefinitions are also included (see Section 5.3).

5.1 Overview

Relatively few works have been published on the topic of recognizing design patterns in C++ source code, and we have found even fewer papers with concrete results.

Kraemer et al. [43] used the Pat system that was developed by Computec GmbH in cooperation with the University of Karlsruhe in Germany. It processes the output of the Paradigm Plus ooCASE tool [1], which is converted to PROLOG facts. It provides a structural analysis of the code based on C++ header files. The relevant extracted information are class names, attribute names, method names and properties, inheritance relations, association and aggregation relations. Some relevant types of information are not extracted by Paradigm, such as the category of a class (abstract or concrete; all classes are considered concrete), the semantic kind of a method (constructor, destructor, etc.) and delegation of method calls (not visible from header files). They found true instances of Adapters and Bridges, but also found false instances of Adapters, Bridges, Composites, Decorators and Proxies. Their precision was between 14-50%.

They examined four real-life projects: the Network Management Environment Browser (NME), the Library of Efficient Datatypes and Algorithms (LEDA), the zApp class library and Automatic Call Distribution (ACD). None of these four benchmarks included explicit design information; all data was extracted from C++ header files, as described above. The structural analysis and the conversion to PROLOG facts took about two hours, while the actual search for the patterns took only a few seconds. All real pattern occurrences were found although the pattern rules could have overlooked some pattern instances because the structural analysis might have mistaken some aggregations (implemented by pointers) for associations. However, it was verified that, with the four benchmarks, none of these cases would have revealed another correct pattern instance. This seemed to be a result of good programming style.

The second work [33] describes a method based on a multi-stage reduction strategy using software metrics and structural properties to extract structural design patterns from an object-oriented design model or source code. Code and design are mapped to an intermediate representation called the Abstract Object Language (AOL). To support

the first case (finding patterns in design) an extractor module called CASE2AOL was implemented for the StP/OMT CASE tool to obtain an AOL specification of the internal object models of the case tool repository. In this case the information extracted is completely trustworthy, in the sense that it really represents design information and no assumptions have to be made about the validity of class relationships. To extract the AOL representation from source code the Code2AOL extractor module was developed for the C++ language. Extracting information about class relationships from code is much harder than extracting information from the design and the result may have some degree of imprecision.

The authors found true instances of Adapters. They also found false instances of Adapters, Bridges and Proxies. The testing included three design patterns: Adapter, Bridge and Proxy. Six public domain systems were analyzed: LEDA, galib, groff, libg++, mec and socket. The first two stages were executed together (metric-based filtering, structural filtering). The third stage (delegation filtering) was executed separately to compare intermediate results. The first stage reduced the input by three to four orders of magnitude. The second stage gave a reduction of one-two orders of magnitude, while the third stage reduced the input by a factor of two to three. The precision after the first two steps was about 55%, and an improvement of 35% was obtained using the delegation constraint making use of structural constraints alone. The correctness was 100% because with their conservative approach no correct instances were missed.

DP++ [4] detects most of the *structural patterns* based on the detection of structural relationships. It also identifies clusters of functionally related classes, which may not necessarily represent any known pattern, but do represent an abstraction in the program. DP++ successfully identified patterns in several commercial and public-domain object-oriented packages, ranging in size from 30 to 400 classes.

PTIDEJ [36] (Pattern Trace Identification, Detection and Enhancement for Java) was developed to perform a search using a constraint satisfaction problem (CSP). The authors analyzed the Java AWT and net libraries and found occurrences of Composite and Facade design patterns.

Brown [10] developed a method for detecting design patterns in SmallTalk. He encoded the pattern detection methods for Composite, Decorator, Template Method and Chain of Responsibility into his algorithm. He then carried out tests on four projects in which he found pattern instances.

5.2 Integration of Columbus and Maisa

In this section we present an approach for recognizing design patterns in C++ source code with the integration of Columbus and Maisa [27]. The method combines the fact extraction capabilities of the Columbus reverse engineering framework with the clause-based pattern mining ability of Maisa.

The Columbus-Maisa pair can be also used for documenting and analyzing software systems implemented in C++. In addition to this, since the major application area of Maisa is the software design phase and that for 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 allows us to scrutinize the software development process.

The method can be used for code comprehension purposes as well to study the structure, behavior and quality of the code. In addition it can be used for tracking the evolution of design decisions between the architectural level and the implementation level of a software system written in C++.

5.2.1 Maisa

Maisa [52; 55] is a software tool for the analysis of software architectures, developed in a research project at the University of Helsinki. The real purpose of 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* [15], Maisa looks for instances of design patterns (either generic ones such as the well-known GoF patterns [34] or user-defined special ones) from the UML diagrams representing the software architecture. Based on what is known 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 the results will be. Therefore design pattern mining at the detailed level of source code – as presented in this section – is the most promising way of improving the practical usability of Maisa.

5.2.2 Integration details

As mentioned in Section 3.1, the facts extracted from a C++ program by Columbus can be accessed through an application programming interface. This API links directly to the schema instance containing the extracted facts about the analyzed project, which is the common internal representation for all the information generated by the C++ extractor. This way, it is quite straightforward to create an exporter plug-in for Columbus that can convert the schema instance into any desired data format.

Because Maisa is implemented entirely in Java, it cannot access the schema instance directly in memory, so we have instead chosen a trivial way of connecting the two tools: an exporter plug-in in Columbus creates a file in Maisa's input file format, which can then be loaded and further processed by Maisa.

The file created by Columbus contains the necessary reverse engineered information in a PROLOG-like format, stored as facts about the main program elements (classes, attributes, etc.) and their relationships (subclassing, composition, etc.). This information

is detailed enough to support, say, the automatic recognition of design patterns from the underlying C++ source code.

5.2.3 Experiments

The design pattern recognition approach outlined above was tested with simple experiments. For this purpose we implemented some of the standard design patterns in C++. After that we used Columbus to analyze the code and to extract high-level structural information from it and to convert it to the input format of Maisa. Finally, Maisa was applied to recognize design patterns in the structural information (and, indirectly, in the original C++ code).

To illustrate this process we will take the *Singleton* [34] design pattern as a simple example. The aim of this pattern is to ensure that a class has only one object 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 aim of Singleton is implemented by a static field that refers to the only instance of the class. The constructor of this class is not accessible to 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 about class relations, such as generalizations, aggregations, associations, as well as the calling dependencies. This information is generated as output by Columbus and put 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").
```

Maisa defines the *Singleton* pattern candidates by the following facts:

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

This description says 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 assignments in the last two lines give the bindings generated by the pattern miner algorithm, with the name of the pattern role on the left-hand side, and the class, attribute, or method having that role in the C++ code on the right hand side.

Table 5.1 summarizes the findings of our experiments. The table gives the names and brief descriptions of the design patterns that were recognized by the Columbus-Maisa pair. While our initial 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.

Pattern name	Description
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
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
Memento	Captures the state of an object

Table 5.1: Design patterns recognized by the Columbus-Maisa pair.

5.3 Pattern miner algorithm in Columbus

Most of the existing approaches for recognizing design patterns in source code only search for *basic pattern structures*. We developed a new method which overcomes this problem by using as much useful information from the source as possible [3]. First, we analyze the C++ source code with the Columbus framework, which then builds the appropriate schema instance. Next, we load our pattern descriptions which are stored in *DPML* (*Design Pattern Markup Language*) files, a new XML-based language that we designed especially for this purpose. These pattern descriptions are easy to modify so as to suit the needs of the users. Finally, our algorithm binds classes found in the source code to pattern classes that are part of the pattern description and checks whether they are related in the way that is described in the pattern. Here we use composition, aggregation, association and inheritance relationships for classes, and call delegation, object creation and operation redefinition (overriding) relationships for operations. The results of the function-body analysis is what gives us more precision compared to other approaches in detecting design pattern occurrences in the source code.

Our system offers users methods for defining their patterns in a very precise way. This means that one can define patterns in the sense of functionality. By doing this, only the pattern instances that fulfill these fine-grained requirements will be identified. This precise definition is, unfortunately, not applicable for every design pattern. Some patterns are so general that they can not even be described in this way, like the *Facade* pattern which defines a higher-level interface to a set of interfaces.

Design pattern mining is a process where the structure of a design pattern is searched for in the source code. The description of the structure should include the main properties of the design pattern and it should at the same time be flexible enough to describe slightly distorted occurrences as well because, in real-world systems, patterns are usually modified to serve the solution of the problem. Because of this the description should be easy to modify and to adapt to the needs of the user. To meet this requirement, we employ our new DPML language to describe design patterns. The language is easy to understand, and the descriptions are readily modifiable.

The search for patterns is performed by trying to match classes found in the source

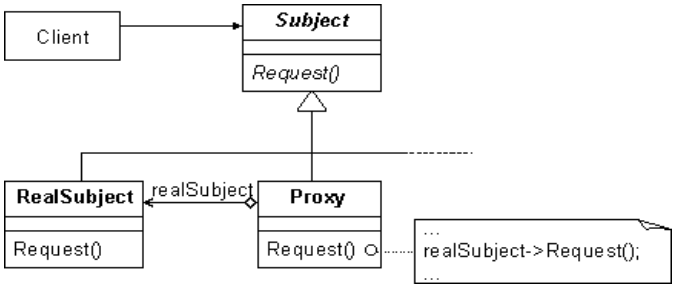


Figure 5.1: The Proxy design pattern

code to classes in the pattern description. Our method shows which class, operation and data member in the source code plays the role of which class, operation and data member in the design pattern, respectively, so it is easy to locate them in the source code.

5.3.1 Design Pattern Markup Language – DPML

Our algorithm uses an XML-based language for design pattern description. This, as we have said, is called the *Design Pattern Markup Language* or *DPML* for short (see Appendix B). We will illustrate the grammar of DPML with an example, that of the description of the Proxy pattern (see Figures 5.1 and 5.2).

The root element is the *<DesignPattern name='...'>* element. Within this element are the classes of the pattern and the type representations. Classes are represented by the *<Class id='id...' name='...' isAbstract='...' isChangeable='...'>* element. The *id* and *name* properties are required, while *isAbstract* and *isChangeable* are optional. The last property tells us that the class can have multiple specimens in a pattern instance. There is no limit to the number of classes. In our example the root element of the pattern description is the DesignPattern element in line 4, which stores the name of the pattern.

The specification of the classes starts with their relations. *<Composition ref='id...' accessibility='...' multiplicity='...'/>* is the element for the composition relationship. The *accessibility* and *multiplicity* properties can be omitted. The other three relations have the same form: *<Aggregation ...>*, *<Association ...>* and *<Base ...>*. When specifying the *Base* (inheritance relation), it makes no sense to specify the multiplicity as it is always 1. In our example, there are class definitions in lines 6, 13 and 27.

The operations of a class can be defined by the *<Operation id='id...' name='...' accessibility='...' storageClass='...' kind='...' isVirtual='...' isPureVirtual='...'>* element. The *id* and *name* properties are required, but the others can be omitted. It may have a *<defines ref='id...'/>* child element to specify which operation it redefines (overrides). The *ref* property is required to be set. Additional child elements may be *<calls ref='id...'>* and *<creates='id...'>* for describing which operations it calls and

```
01<?xml version='1.0'?>
02<!DOCTYPE DesignPattern SYSTEM 'dpml-1.6.dtd'>
03
04<DesignPattern name='Proxy'>
05
06  <Class id='id10' name='Subject' isAbstract='true'>
07    <Operation id='id11' name='Request' kind='normal'
08               isVirtual='true' isPureVirtual='true'>
09      <hasTypeRep ref='id50'/>
10    </Operation>
11  </Class>
12
13  <Class id='id20' name='Proxy'>
14    <Base ref='id10'/>
15    <Aggregation ref='id30'/>
16    <Operation id='id21' name='Request' kind='normal'
17               isVirtual='true' isPureVirtual='false'>
18      <defines ref='id11'/>
19      <calls ref='id31'/>
20      <hasTypeRep ref='id50'/>
21    </Operation>
22    <Attribute id='id22' name='realSubject'>
23      <hasTypeRep ref='id52'/>
24    </Attribute>
25  </Class>
26
27  <Class id='id30' name='RealSubject'>
28    <Base ref='id10'/>
29    <Operation id='id31' name='Request' kind='normal'
30               isVirtual='true' isPureVirtual='false'>
31      <defines ref='id11'/>
32      <hasTypeRep ref='id50'/>
33    </Operation>
34  </Class>
35
36  <TypeRep id='id50'>
37    <TypeFormerFunc>
38      <hasReturnTypeRep ref='id51'/>
39    </TypeFormerFunc>
40  </TypeRep>
41
42  <TypeRep id='id51'/>
43
44  <TypeRep id='id52'>
45    <TypeFormerPtr/>
46    <TypeFormerType ref='id30'/>
47  </TypeRep>
48
49</DesignPattern>
```

Figure 5.2: The Proxy pattern in DPML

which objects it creates, respectively. The *ref* property is required here. The operation has a *<hasTypeRep ref='id...'/>* child element which refers to its type representation

(see below). The *ref* property is required here. The operation can have parameters that are specified by the <*Parameter id='id...' name='...'*> element. Both properties are needed. The parameter element also has a <*hasTypeRep ref='id...'/*> child element which refers to the type representation of the parameter.

Class attributes can be defined by the <*Attribute id='id...' name='...' accessibility='...' storageClass='...'*> element. The *id* and *name* properties are required, but the other two can be omitted. A class attribute – like an operation – has a <*hasTypeRep ref='id...'/*> child element which refers to its type representation (see below).

In our example the first class (lines 6-11) called *Subject* is abstract, and has an operation *Request* which is pure virtual. Its type (line 9) is represented by TypeRep id50 (line 36). The second class *Proxy* (lines 13-25) is derived (line 14) from class *Subject*–id10 and aggregates the class *RealSubject*–id30 (line 15). It also has an operation *Request* (lines 16-21) which is virtual, but not pure virtual. It defines/overrides the inherited operation *Request*–id11 (line 18) and calls the operation *Request*–id31 from class *RealSubject* (line 19). Its type is represented by TypeRep id50 (line 20). This class has an attribute called *realSubject* as well (lines 22-24) whose type is represented by TypeRep id52 (line 23). The third class *RealSubject* (lines 27-34) is derived (line 28) from the class *Subject*–id10 too. It also has an operation *Request* (lines 29-33) which is virtual but not pure virtual. It also defines/overrides the inherited operation *Request*–id11 (line 31). Its type is represented by TypeRep id50 (line 32).

The <*TypeRep id='id...'*> element represents a type. It can have different child elements which modify the referred type, like <*TypeFormerArr/*> for arrays, <*TypeFormerPtr/*> for pointers/references and <*TypeFormerFunc*> for functions. The reference to the base type is stored in a <*TypeFormerType* ref='id...'/> element. The <*TypeFormerFunc*> element has a <*hasReturnTypeRep ref='id...'/*> child element for referring to the return type and may have several <*hasParameterTypeRep ref='id...'/*> child elements for referring to the types of the parameters.

In the example the first type representation (lines 36-40) shows a function type (line 37) and its return type is described in TypeRep id51 (line 38). The second type representation (line 42) is empty, which means that it can represent any type. The third type representation (lines 44-47) means a pointer (line 45) to an object of type id30–*RealSubject* (line 46).

5.3.2 The pattern miner algorithm

First, the Columbus framework analyzes the C++ source code and builds a schema instance – abstract semantic graph (ASG) – from it. Second, the appropriate DPML pattern description file is loaded into a standard XML DOM tree. The pattern miner algorithm (see Figure 5.3) then matches the DOM tree to the ASG. This process is similar to graph matching where the vertices are classes and the edges are relations among the classes.

Columbus also calculates the class diagram and call graph from the ASG. Since class diagrams are reconstructed from source code in different ways by different reverse engineering tools, we will explain here what we mean under the concept of inheritance, composition, aggregation and association. *Inheritance* means that a class derives from another class, called the base class (for design pattern detection, inheritances are stored transitively as well). *Composition* means that a class contains another class instance by a data member. *Aggregation* means that a class contains another class by having a pointer or a reference to it. *Association* means that a class uses another class as an operation parameter's type or as an operation's return type. The class diagram and call graph are used later in the algorithm.

The algorithm starts by collecting *source class*[1] candidates for each *pattern class*[2]. This is accomplished by searching for classes which have adequate attributes and operations with the desired properties[3]. Only the properties given in the pattern description are checked. The existence of the required relations[4] of the class is checked as well. If a source class has all the needed attributes, operations and relations it is then stored as a candidate for the appropriate pattern class. One source class can become a candidate for multiple pattern classes.

In the second stage the candidates are filtered. This is done by checking the connections with other pattern class candidates: if two pattern classes are related in some way, then all candidate classes of the first pattern class have to be connected in the same way to at least one candidate class of the second pattern class. If a class does not have the necessary relations, it is removed from the candidates list. This step is repeated iteratively until no further classes are removed.

Next, all combinations of the candidate classes have to be tested to find design pattern instances. The algorithm searches for these combinations recursively to check every possibility (procedure *bindSourceClassToPatternClass*). If a combination of classes has all the required connections, the attributes and operations of the source classes have to be matched to the attributes and operations of the pattern classes.

These attributes and operations are also matched recursively to try out every combination (function *bindAttributesAndOperations*). A source class attribute is bound to a pattern class attribute if it has all the required properties, checking this time its type as well. For operations, the return types and parameters have to be checked too.

After a combination is found, a further check has to be performed to see if the bound functions contain the needed call delegations, object creations and whether they redefine/override the appropriate operations (function *checkCallsCreatesDefines*). This is performed sequentially for each class and for each function.

If this last check was successful, then the current bound source classes form a design

[1] A class found in the source code.
[2] A class found in the pattern description.
[3] Simple properties like virtuality and accessibility. There is no type-checking at this stage.
[4] Relations can be inheritance, composition, aggregation and association.

pattern instance. These classes, their path and line information and the role they play in the design pattern instance are then displayed on the screen (together with the bound operations and attributes).

5.3.3 Algorithm optimizations

The algorithm outlined above roughly describes how our algorithm works. Of course, the basic algorithm is not optimal, so we will present two optimizations that improve its performance.

The first optimization (see Figure 5.4) further reduces the candidate lists using the information provided by the class diagram built at the beginning. The problem is the following: let us take a big system with about 10000 classes. An average pattern consists of 3 classes. Let us presume that for each class there are about 1000 candidates after the filtering. Then about 1000*1000*1000 = 10^9 combinations have to be checked, and this is too expensive.

This number can be dramatically reduced by using the following technique. First, a graph is built where the vertices are the pattern classes and the edges are the relations among them. Second, a topological ordering is performed on it. In this way an ordering of pattern classes is carried out where the latter ones depend on a previous one. The procedure *makeSmallCandidateSets* creates new smaller candidate class subsets for a given pattern class from the original candidates set. These will take the place of the original (large) candidate sets in the *bindSourceClassToPatternClass* procedure. The union of these small sets forms the original set.

The small sets are created in the following way. The first relation is considered between two pattern classes where the second class depends on the first one (the first pattern class occurs before the second one in the topological order). For each candidate class of the first pattern class, the small set consists of those candidate classes of the second pattern class which are related in the same way to the first candidate class as the second pattern class is related to the first one. Each pattern class can play the role of the second pattern class in the previous discussion only once. This way a small set is unambiguously defined by two pattern classes and a source class that plays the role of the first pattern class.

These small sets are used in procedure *bindSourceClassToPatternClass* in the following way. If during class binding previously created small sets exist which are defined by the actual pattern class, the source class bound to it and the dependent pattern classes, then the candidate sets of the dependent ones are replaced by the small ones. If the small sets are 10 times smaller than the original ones, say, then the number of combinations will be 1000*100*100 = 10^7, which is an improvement of two orders of magnitude (this, of course, strongly depends on the system being analyzed).

The second optimization is a cut in the operation and attribute matching. If a pattern element is considered which says that an operation must create an object, then it

Size info	Jikes	LEDA	Calc	Writer
No. of files	77	488	6 307	6 794
Size	2.7MB	2.7MB	44.5MB	52.2MB
LOC[5]	75 485	85 606	1 270 303	1 512 972
No. of classes	329	1 617	5 051	6 729

Table 5.2: Information about the size of the projects.

can be checked whether the operation being matched is indeed creating an object of the given type. If it does not, this combination of operations and attributes is not adequate and the matching will continue with another source operation. In the case of call delegation and operation redefinition this checking is possible only if the called or redefined operation already has a bound source class element. Specifically (see Figure 5.5), this means that the *bindAttributesAndOperations* function is modified so that it calls a new *checkOperationConstraints* function that checks the above described conditions.

5.3.4 Experiments

In this section we demonstrate the design pattern detection capabilities of our system. The experiments were performed on four real-life, publicly available C++ projects (Table 5.2 summarizes their size information) listed below:

- *Jikes* [42]. Open source Java compiler system from IBM.
- *LEDA* [44; 49]. Library of efficient data types and algorithms.
- *StarOffice Calc* [62]. The spreadsheet application of StarOffice, a large C++ project that consist of 6,307 source files (more than 1.2 million non-preprocessed non-empty lines of code).
- *StarOffice Writer* [62]. The word processing application of StarOffice, a large C++ project that consist of 6,794 source files (more than 1.5 million non-preprocessed non-empty lines of code).

Table 5.3 shows the number of different design pattern instances found in the test projects. For most design patterns we also wrote their "soft" versions in which we slightly relaxed the original specifications in [34] (for instance, we did not demand that some classes be abstract). The patterns are grouped just as in [34]: creational, structural and behavioral. Except for LEDA, the other three are more recent projects, and it was noticed that there were many more patterns in these projects than in LEDA.

Table 5.4 shows the percentage of true pattern instances among the ones that were found. All found instances (except the instances of *Adapter Object* and its soft version)

[5]Lines of non-empty non-preprocessed code.

Statistics	Jikes	LEDA	Calc	Writer
Abstract Factory	-	-	-	-
Builder	-	-	2	7
Builder soft	-	-	17	9
Factory Method	-	-	-	-
Factory Method soft	-	-	1	9
Prototype	1	-	-	1
Prototype soft	1	-	-	1
Singleton	-	-	-	-
Adapter Class	-	-	-	16
Adapter Class soft	-	-	13	16
Adapter Object	54	-	27	62
Adapter Object soft	62	-	153	135
Bridge	-	-	-	-
Bridge soft	-	-	73	80
Decorator	-	-	-	-
Decorator soft	-	-	-	-
Proxy	36	-	-	4
Proxy soft	44	-	-	5
Chain of Responsibility	-	-	-	-
Iterator	-	-	-	-
Iterator soft	-	-	1	-
Strategy	4	1	10	5
Strategy soft	12	2	20	32
Template Method	5	-	94	101
Visitor	-	-	-	-
Visitor soft	-	-	-	5
Sum total	235	6	442	525

Table 5.3: Number of design pattern instances found.

were checked manually in the source code (design patterns for which no instances were found are not shown). Table 5.5 shows the pattern mining time[6] for different patterns (it does not include the time needed for source code analysis). LEDA and Jikes are medium-sized projects where the search took only a few minutes, which is acceptable. StarWriter and StarCalc are huge projects where the search took several hours. This is reasonable if we consider that they each contain more than a million lines of code, but the time also indicates that we should try even more to optimize our algorithm. The search time is long mainly in cases when lots of operations and attributes in the source classes need to be bound to pattern operations and attributes. This part of the algorithm should be optimized.

The testing we performed revealed several things. Firstly, the larger a project is, the more likely it will contain design patterns. Since the patterns are implemented to solve a specific problem in a specific environment, they rarely follow the strict descriptions in [34]. The implementations usually violate some rules, so the pattern description

[6]All tests were performed on an Intel P4-2000 machine with 512MB RAM.

Statistics	Jikes	LEDA	Calc	Writer
Builder	-	-	0%	14%
Builder soft	-	-	0%	11%
Factory Method soft	-	-	100%	100%
Prototype	100%	-	-	100%
Prototype soft	100%	-	-	100%
Adapter Class	-	-	-	0%
Adapter Class soft	-	-	0%	0%
Bridge soft	-	-	100%	100%
Proxy	0%	-	-	50%
Proxy soft	18%	-	-	60%
Iterator soft	-	-	0%	-
Strategy	100%	100%	100%	100%
Strategy soft	100%	100%	100%	100%
Template Method	100%	-	100%	100%
Visitor soft	-	-	-	0%

Table 5.4: Percentage of true pattern instances.

must be simplified to be able to recognize them. But the more we simplify the pattern description, the more false positives we will get. Sometimes it is almost impossible to determine whether a pattern instance we have found is a real or false instance. The structure may be proper, but the pattern instance does not do what we expect. This means that we can find source classes that are connected in the right way, but we cannot check whether they fulfill the aim of the original pattern. This can be checked only manually.

The names of the operations and classes usually suggest the role they play, like the operation *Clone* in a *Prototype* instance found in *Jikes*, or *CreateNew* in a *Prototype* found in *StarWriter*. As before, some names indicated that it was a false positive we were looking at, but we could not eliminate them. These false positives also have the required structure, but their aim is different from the design pattern they are similar to. As for trivial patterns like the *Template Method*, there is no possibility of finding false positives. On the other hand, all the instances of *Adapter Class* are false positives, because we could not describe the delegation of the important part of the job, so we obtained results where only marginal parts of the jobs were delegated to the *Adaptee* class. In the case of the *Proxy* pattern where we found true and false instances too, we accepted only those instances where the call delegation did the main part of the job.

We encountered an interesting problem during the pattern formalization process. The structure of the *State* and *Strategy* patterns are so similar that we could not distinguish them in the pattern description. The differences are in motivation and intent, which we could not formalize. That is why the tables do not contain the results for the *State* pattern.

Time (h:mm:ss)	Jikes	LEDA	Calc	Writer
Abstract Factory	0:00:00	0:00:00	0:00:02	0:00:10
Builder	0:00:01	0:00:00	0:00:11	0:00:29
Builder soft	0:00:02	0:00:02	0:32:41	0:47:05
Factory Method	0:00:00	0:00:00	0:00:02	0:00:12
Factory Method soft	0:00:00	0:00:00	0:02:00	0:04:06
Prototype	0:00:43	0:00:00	0:00:22	0:00:47
Prototype soft	0:00:44	0:00:00	0:14:57	0:30:12
Singleton	0:00:00	0:00:00	0:00:00	0:00:00
Adapter Class	0:00:00	0:00:00	0:00:01	0:00:01
Adapter Class soft	0:00:00	0:00:00	0:00:01	0:00:16
Adapter Object	0:00:01	0:00:00	0:01:47	0:02:12
Adapter Object soft	0:00:04	0:00:00	0:20:31	0:30:22
Bridge	0:00:00	0:00:00	0:00:00	0:00:23
Bridge soft	0:00:00	0:00:01	0:49:19	1:53:08
Decorator	0:00:00	0:00:00	0:00:00	0:00:01
Decorator soft	0:00:00	0:00:00	0:05:05	0:11:56
Proxy	0:00:00	0:00:00	0:00:00	0:00:00
Proxy soft	0:00:00	0:00:00	0:00:01	0:00:02
Chain of Resp.	0:00:00	0:00:00	0:00:00	0:00:01
Iterator	0:00:00	0:00:00	0:00:00	0:00:00
Iterator soft	0:00:00	0:00:00	0:00:00	0:00:00
Strategy	0:00:33	0:00:00	0:00:42	0:01:34
Strategy soft	0:00:38	0:00:30	1:27:01	2:27:56
Template Method	0:00:10	0:00:00	0:00:11	0:00:51
Visitor	0:00:00	0:00:00	0:00:00	0:00:01
Visitor soft	0:00:00	0:00:00	0:00:01	0:00:04
Sum total	0:04:08	0:01:04	5:02:47	9:00:53

Table 5.5: Pattern mining time.

5.4 Summary

In this chapter we introduced two methods for discovering design pattern instances in C++ source code.

First, we presented a method and tool set for recognizing design patterns with the integration of Columbus and Maisa. The method combines the fact extraction capabilities of the Columbus reverse engineering framework with the clause-based pattern mining ability of Maisa. The C++ code was analyzed by Columbus, and a plug-in was written to export the collected facts to a form understandable to Maisa. This form is a clause-based design notation in PROLOG. Afterwards, this file was analyzed by Maisa, and those pattern instances that match the previously given design pattern descriptions were searched for. No real-world projects were analyzed, but the reference implementations of seven different design patterns were identified.

Second, we showed a new approach to the problem of pattern detection which includes the detection of call delegations, object creations and operation redefinitions. These are the elements that identify pattern occurrences more precisely. The pattern

descriptions are stored in our new XML-based format, the Design Pattern Markup Language (DPML). This gives the user the freedom to modify the patterns, adapt them to his or her own needs, or create new pattern descriptions. We then tested our method on four public-domain projects. The main advantage that this work offers is that it facilitates design pattern instance detection by using a user-friendly language for design pattern description. Thus it will be straightforward to identify pattern instances with this tool, and it can be helpful in code comprehension, code documentation, correct pattern implementation testing, and in other fields which require pattern mining.

```
function checkCallsCreatesOverloadsDefines() : bool
  foreach pattern class P1 ∈ PC
    foreach operation O1 ∈ O(P1)
      foreach relation R between operations O1 and O2
        if R ∉ RO(actBoundElement(O1),actBoundElement(O2)) then
          return false
      foreach relation R between operation O1 and pattern class P2
        if R ∉ ROC(actBoundElement(O1),actBoundClass(P2)) then
          return false
  return true
endfunc

function bindAttributesAndOperations(PC index i, PE index j) : bool
  if j ≤ PC[i].size then
    foreach element E ∈ actBoundClass(PC[i])
      if E is not yet bound then
        if P(PE(PC[i],j)) ⊆ P(E) and Type(PE(PC[i],j)=Type(E) then
          actBoundElement(PE(PC[i],j)) := E
          if bindAttributesAndOperations(i,j+1) then return true
          clear binding of PE(PC[i],j)
  else
    if i ≤ PC.length then
      if bindAttributesAndOperations(i+1,1) then return true
    else
      if checkCallsCreatesDefines() then return true
  return false
endfunc

procedure bindSourceClassToPatternClass(PC index i)
  if i ≤ PC.length then
    foreach C ∈ Candidates(PC[i])
      if C is not yet bound then actBoundClass(PC[i]) := C
      bindSourceClassToPatternClass(i+1)
  else
    foreach pattern class P1 ∈ PC
      foreach relation R between pattern classes P1 and P2
        if rel(R,actBoundClass(P1),actBoundClass(P2)) then
          if bindAttributesAndOperations(1,1) then
            the bound source classes form a design pattern instance
    clear all bindings for operations and attributes
endproc

program PatternMiner
  analyze the subject system and build the schema instance
  calculate the class diagram
  calculate the call graph
  load pattern classes from DPML to PC
  foreach pattern class P ∈ PC
    foreach source class S
      if P(P) ⊆ P(S) and A(P) ⊆ A(S) and O(P) ⊆ O(S) and R(P) ⊆ R(S) then
        Candidates(P) += S
  repeat
    foreach pattern class P1 ∈ PC
      foreach relation R between pattern classes P1 and P2
        foreach candidate class C ∈ Candidates(P1)
          if not relSet(R,C,Candidates(P2)) then
            Candidates(P) -= C
  until no further classes are removed
  bindSourceClassToPatternClass(1)
endprog

PC – list of pattern classes
A(C) – set of attributes of class C
O(C) – set of operations of class C
R(C) – set of class-class relations of class C (inheritance, composition, aggregation, association)
RO(F1,F2) – set of operation-operation relations between operations F1 and F2 (call, redefine)
ROC(F,C) – set of operation-class relations between operation F and class C (creates)
Candidates(C) – set of candidate source classes for pattern class C
rel(R,C1,C2) – checks if a relation R exists between classes C1 and C2
relSet(R,C,S) – checks if a relation R exists between class C and at least one of the classes from set S
PE(P,i) – i. element (operation or attribute) of the P pattern class
actBoundClass(P) – source class bound to the P pattern class
actBoundElement(E) – source class element bound to pattern class element E
P(X) – set of properties of X (X can be a class or class element)
Type(E) – type of class element E
```

Figure 5.3: The pattern miner algorithm

```
procedure makeSmallCandidateSets()
  perform topological ordering of the pattern classes and store them ordered to TO
  foreach pattern class P1 ∈ TO
    Visited(P1) := true
    foreach relation R between pattern classes P1 and P2
      if not Visited(P2) then
        Visited(P2) := true
        foreach candidate class C1 ∈ Candidates(P1)
          foreach candidate class C2 ∈ Candidates(P2)
            if rel(R,C1,C2) then
              Candidates(P1,C1,P2) += C2
endproc

procedure bindSourceClassToPatternClass(PC index i)
  if i ≤ PC.length then
    foreach C ∈ Candidates(PC[i])
      if C is not yet bound then actBoundClass(PC[i]) := C
→      foreach PC[j]: j>i and Candidates(PC[i],C,PC[j])≠∅
→        Candidates(PC[j]) := Candidates(PC[i],C,PC[j])
      bindSourceClassToPatternClass(i+1)
  else
    ...
endproc

program PatternMiner
  ...
  until no further classes are removed
→makeSmallCandidateSets()
→PC:=TO
  bindSourceClassToPatternClass(1)
endprog

TO – topologically sorted list of pattern classes
Candidates(P1,C,P2) – set of candidate source classes for pattern
                      class P2 if C plays the role of P1
```

Figure 5.4: Candidates lists reduction

```
function checkOperationConstraints(pattern element O1) : bool
  if O1 is not an operation then return true
  foreach relation R between operations O1 and O2
    if actBoundElement(O2)=∅ then
      return true
    if R ∉ RO(actBoundElement(O1),actBoundElement(O2)) then
      return false
  foreach relation R between operation O1 and pattern class P2
    if R ∉ ROC(actBoundElement(O1),actBoundClass(P2)) then
      return false
  return true
endfunc

function bindAttributesAndOperations(PC index i, PE index j) : bool
  if j ≤ PC[i].length then
    foreach element E ∈ actBoundClass(PC[i])
      if E is not yet bound then
        if P(PE(PC[i],j)) ⊆ P(E) and Type(PE(PC[i],j)=Type(E) then
          actBoundElement(PE(PC[i],j)) := E
→           if checkOperationConstraints(PE(PC[i],j)) then
              if bindAttributesAndOperations(i,j+1) then return true
            clear binding of PE(PC[i],j)
  else
    ...
endfunc
```

Figure 5.5: Cut in the operation and attribute matching

"Any clod can have the facts; having opinions is an art."
Charles McCabe

Chapter 6

Analyzing the fault-proneness of open source software

Open source software systems are becoming evermore important these days. Many large companies are investing in open source projects and plenty of them are also using this kind of software in their own work. As a consequence, many of these projects are being developed swiftly and soon become very large. But because open source software is usually developed outside companies – mostly by volunteers – the quality and reliability of the code may be uncertain. Thus its use may involve big risks for a company that uses open source code. It is vital to get as much information out of these systems as possible.

Various kinds of code measurements can be quite helpful in obtaining information about the quality and fault-proneness of the code. These measurements can be done with the help of proper software tools.

In this chapter we describe how we calculated the object-oriented metrics validated in [5], [7], [8] and [9] for fault-proneness detection (similar results were also presented in [32] and [67]) from the source code of the well-known open source web and e-mail suite called *Mozilla* [50; 56]. We then compare our results with those presented in [5]. One of our aims was to supplement their work with metrics obtained from a real-world software system.

We also compare the metrics of seven versions of Mozilla (1.0–1.6), which covers more than one and a half years of development, to see how the predicted fault-proneness of the software system changed during its development cycle [29].

It should be mentioned here that we performed complete analyses of the seven versions of Mozilla and built up the full schema instances of them, which can be used for any re- and reverse engineering task like architecture recovery and visualization. In this chapter we just used them for calculating metrics.

6.1 Overview

Mozilla was investigated earlier by Godfrey and Lee [35]. They examined the software architecture model of Mozilla Milestone-9. The authors used the *PBS* [31] and *Acacia* [13] reverse engineering systems for fact extraction and visualization. They created the subsystem hierarchy of Mozilla and looked at the relationships among them. Their model consists of 11 top-level systems which may be divided into smaller subsystems. They created the subsystem hierarchy by taking into consideration things like source directory structure and software documentation. It turned out that the dependency graph was near complete, which means that almost all the top-level systems used each other.

Fioravanti and Nesi [32] took the results of the same projects as Basili *et al.* in [5] to examine how metrics could be used for fault-proneness detection. They calculated 226 metrics and their aim was to choose a minimal number relevant for obtaining a good identification of faulty classes in medium-sized projects. First they reduced the number of metrics to 42 and attained a very high accuracy score (more than 97%). This model was still too large to be useful in practice. By using statistical techniques based on logistic regression they created a hybrid model which consists of only 12 metrics with an accuracy that is still good enough to be useful (close to 85%). The metrics suite they obtained is not the same as the one used in [5] but there are many similarities.

Yu, Systä and Müller chose eight metrics in [67] (actually ten because CBO and RFC were divided into two different kinds) and they examined the relationship between these metrics and the fault-proneness of the software. The subject system was the client side of a network service management system developed by three professional software engineers. It was written in Java and consisted of 123 classes and around 34,000 lines of code. First they examined the correlation among the metrics and found four highly correlated subsets. Then they used univariate analysis to discover which metrics could detect faults and which could not. They found that three of the metrics (CBO_{in}, RFC_{in} and DIT) were unimportant while the others were significant but to different extents (NMC, LOC, CBO_{out}, RFC_{out}, LCOM, NOC and Fan-in).

6.2 Fact extraction with compiler wrapping

In this section we describe how our fact extraction process (see Section 4.1) works in practice when using the CANGccWrapper toolset. We successfully extracted facts from Mozilla and calculated various metrics from the schema instances that we obtained. As can be seen in Figure 4.1, the process consists of five consecutive steps.

Step 1 (Acquiring project/configuration information) is implicitly handled by our wrapper toolset because it is integrated into the usual build process.

Step 2 (Analysis of the source code) is done using the CANPP and CAN tools, which are invoked by CANGccWrapper when the source files are compiled (as described previously). One of the issues with which we had to deal in this step was that the outputs of CANPP (the preprocessed file and the corresponding schema instance) are not present in the makefiles of the real build process. The first is needed immediately by CAN, so there is no problem in handing it over, but the schema instances are needed later in Step 3 when they must be linked to each other. In that step only the compiled (object) and linked filenames are available in the parameters and we have to find the files containing the associated schema instances. We solved this problem by using the same filename for the outputs but with different extensions. These files are always stored next to the original output (in the same directory). CANGccWrapper first calls CANPP with the appropriate parameters and the input file. CANPP analyzes and preprocesses it and then saves the extracted schema instance to a file with the extension "*.psi*" and the preprocessed file to a file with the extension "*.i*". Afterwards, CANGccWrapper invokes CAN with the appropriate parameters and input/output files. CAN uses the preprocessed file created by CANPP, analyzes it then saves the extracted schema instance to a file with the extension "*.csi*".

In *Step 3 (Linking of schema instances)* we have two kinds of schema instances as inputs. These are linked in parallel with the linking of the regular object files. CANGccWrapper invokes CANLink to perform this step. As a result there will be only a few files which contain all the extracted information that belong to the different subsystems of the subject system. The function of the wrapper ends here in this step, where all the extracted information is available for further processing.

We found some configuring examples in Mozilla's makefiles that seem quite "sneaky." For instance, objects were moved to another directory before linking, so our wrapper did not find the files containing the associated schema instances (it searched for them next to the original location of the object files). Our solution was to wrap the *cp* (file copy), *mv* (file move) and *ln* (file link) commands in a similar way to that done with the compiler.

Step 4 (Filtering the schema instances) deals with filtering the schema instances that were obtained. Since this step is optional (the linked schema instances can be used as they are as well – without any filtering) and we wanted to analyze the whole system, we did no filtering at all.

The extracted information can be used for many purposes. Here we calculated object-oriented metrics in *Step 5 (Processing the schema instances)* from the schema instances with the help of the *CAN2Metrics* command line tool (the output is a simple CSV–Comma Separated Value file which can be easily processed by statistical programs and spreadsheet editors, for instance).

6.3 Experiments

In this section we analyze Mozilla using metrics. We collected metrics from seven different versions of Mozilla from 1.0 released in August of 2002 to 1.6 released in January of 2004. We did not classify the metrics according to their purpose or usability nor did we search for correlations between them. Instead we used the results of Basili *et al.* and studied the metrics according to [5].

Basili *et al.* studied object-oriented systems written by students in C/C++. They carried out an experiment in which they set up eight project groups consisting of three students each. Each group had the same task – to develop a small/medium-sized software system. Since all the necessary documentation (for instance, reports about faults and their fixes) was available, they could search for relationships between the fault density and metrics. They used a metrics suite that consisted of six metrics and analyzed the distribution and correlations between them. Afterwards, they made use of logistic regression – a standard technique based on maximum likelihood estimation – to analyze the relationship between metrics and the fault-proneness of classes. In this chapter we refer to their projects as the *reference project*.

In the following we define the six metrics they investigated.

WMC (Weighted Methods per Class). The WMC is the number of methods defined in each class. More precisely, WMC is defined as being the number of all member functions and operators defined in each class. However, "friend" operators (C++ specific construct) are not counted. Member functions and operators inherited from the ancestors of a class are also not counted.

DIT (Depth of Inheritance Tree). The DIT measures the number of ancestors of a class.

RFC (Response For a Class). This is the number of methods that can potentially be executed in response to a message being received by an object of that class. The RFC is the number of C++ functions directly invoked by member functions or operators of a C++ class.

NOC (Number Of Children). This is the number of direct descendants for each class.

LCOM (Lack of Cohesion on Methods). This is the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. However, the metric is set to zero whenever the above subtraction is negative.

CBO (Coupling Between Object classes). A class is coupled to another one if it uses its member functions and/or instance variables. The CBO gives the number of classes to which a given class is coupled.

We should mention here that some of these metrics have different names in the Columbus framework (for instance DIT is called NOA – Number Of Ancestors) but, for

simplicity, we use their corresponding names in this chapter.

Table 6.1 lists some size metrics of the analyzed versions of Mozilla.

ver.	NCL	TLOC	TNM	TNA
1.0	4,770	1,127,391	69,474	47,428
1.1	4,823	1,145,470	70,247	48,070
1.2	4,686	1,154,685	70,803	46,695
1.3	4,730	1,151,525	70,805	47,012
1.4	4,967	1,171,503	72,096	48,389
1.5	5,007	1,169,537	72,458	47,436
1.6	4,991	1,165,768	72,314	47,608

NCL: Number of Classes.
TLOC: Total number of non-empty lines of code.
TNM: Total Number of Methods in the system.
TNA: Total Number of Attributes in the system.

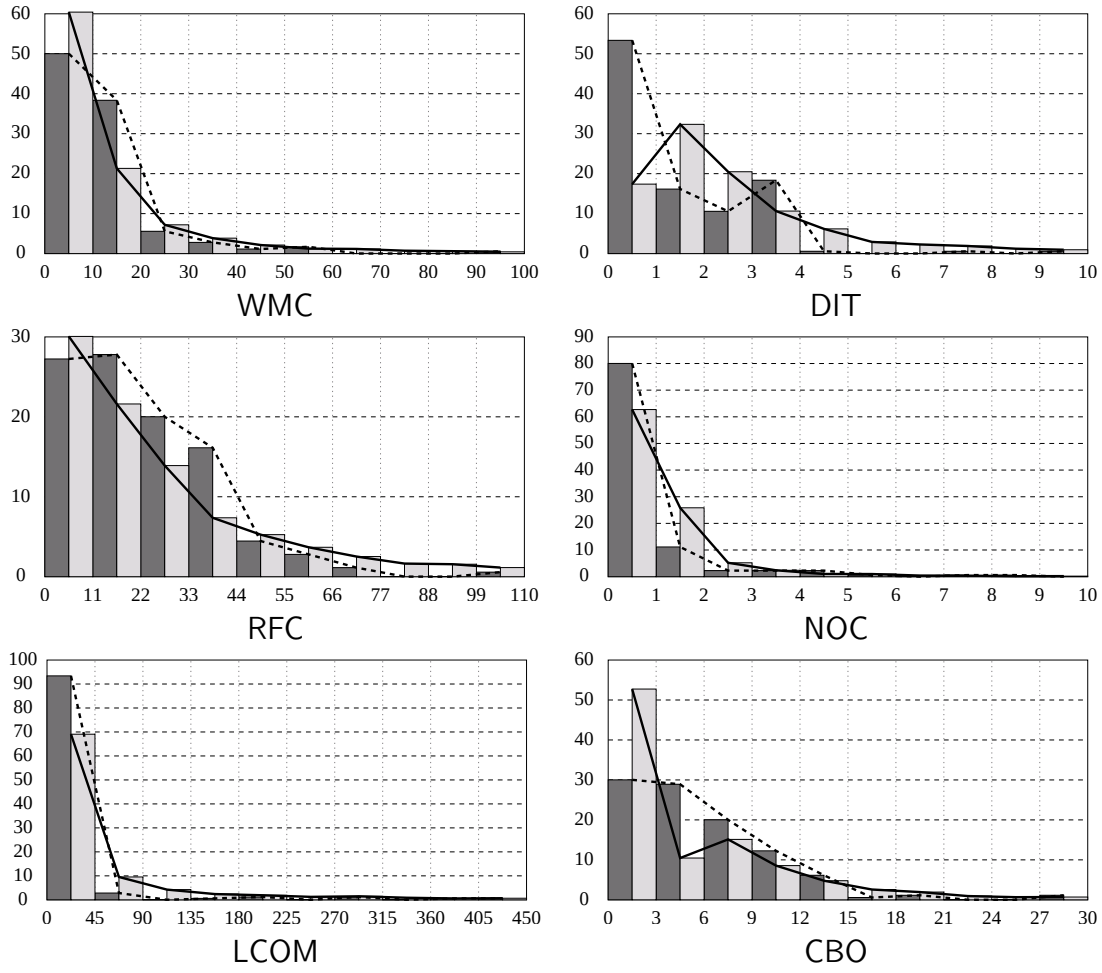
Table 6.1: System-level metrics of the analyzed Mozilla versions.

In the following we compare some statistical data and the distribution of metrics collected from Mozilla and the reference project. Afterwards, using the findings of the reference project in [5], we investigate how Mozilla's predicted fault-proneness changed over a seven-version evolution cycle.

6.4 Comparison of reference project with Mozilla

In this section we compare the metrics calculated for Mozilla 1.6 with those of the reference project. Figure 6.1 shows a comparison of the *distribution of the metrics*. It can be seen that the distributions of WMC, RFC, NOC and LCOM are quite similar in both cases. On the other hand, the distributions of DIT and CBO are quite different.

We also compared the statistical information we obtained from the measurements. Table 6.2 shows basic statistical information about the two systems. The *Minimum* values here are almost the same but the *Maximum* values have increased dramatically. This is not surprising because Mozilla has about 30 times more classes than the reference project. Since LCOM is proportional to the square of the size (number of methods) of a class, its very large value is to be expected. In Mozilla there are about five thousand classes, so the extremely high value of NOC may seem surprising at first. But the second biggest value of NOC is just 115, hence we think that the class with the largest value is probably a common base class from which almost all other classes are inherited. *Median* and *Mean* express "a kind of average" and they are more or less similar, except for the LCOM value (similar to the Maximum value). Since in Mozilla there are many more classes and these are more variegated, the metrics change over a wider range. The *Standard Deviation* values suggest this bigger variety of the classes.



The X axes represent the values of the metrics. The Y axes represent the percentage of the number of classes having the corresponding metrics value.
The darker columns represent the original values of the reference project from [5], while the brighter ones represent the values calculated for Mozilla 1.6.

Figure 6.1: Distribution of the metrics of the reference project and Mozilla.

Basili *et al.* [5] also calculated the *correlations* of the metrics (see Table 6.3), which is also an important statistical quantity. They found that the linear Pearson's correlation (R^2: Coefficient of determination) between the object-oriented metrics studied are, in general, very weak. Three coefficients of determination appear somewhat more significant, but they conclude that these metrics are mostly statistically independent. We also calculated the same correlations in the case of Mozilla 1.6 but we obtained different results. NOC is independent of the other metrics just like in the reference project, but all the others have some correlation with each other. There are three very weak correlations but the rest represent more or less significant correlations. What is more, there are some very large values (for instance, between WMC and LCOM), so it follows that these metrics are not totally independent and represent redundant information. This

Ref. | **Moz.**	WMC		DIT		RFC	
Maximum	99.00	**337.00**	9.00	**33.00**	105.00	**1,074.00**
Minimum	1.00	**0.00**	0.00	**0.00**	0.00	**0.00**
Median	9.50	**7.00**	0.00	**2.00**	19.50	**21.00**
Mean	13.40	**14.12**	1.32	**2.39**	33.91	**48.95**
Standard deviation	14.90	**22.16**	1.99	**2.90**	33.37	**81.99**
Ref. | **Moz.**	NOC		LCOM		CBO	
Maximum	13.00	**1,213.00**	426.00	**55,198.00**	30.00	**70.00**
Minimum	0.00	**0.00**	0.00	**0.00**	0.00	**0.00**
Median	0.00	**0.00**	0.00	**15.00**	5.00	**2.00**
Mean	0.23	**1.06**	9.70	**273.82**	6.80	**5.11**
Standard deviation	1.54	**17.44**	63.77	**1,597.53**	7.56	**7.49**

The bold numbers represent the values of Mozilla 1.6, while the normal ones are the values of the reference project.

Table 6.2: Descriptive statistics of the classes in the reference project and Mozilla.

is surprising because Basili *et al.* found that some of these metrics could be used for detecting fault-proneness while the others were not significant. Unfortunately, we could not verify this for Mozilla, because we did not have such good documentation about the faults they had. (We should mention here that the reported faults about Mozilla are available from Bugzilla [11], but it is difficult to collect and group the necessary data.)

We did not find any information that contradicted [5], so we accepted their hypotheses and results and investigated the changes of Mozilla over its seven-version development.

Reference	WMC	DIT	RFC	NOC	LCOM	CBO
WMC	1	0.02	**0.24**	0	**0.38**	0.13
DIT		1	0	0	0.01	0
RFC			1	0	0.09	**0.31**
NOC				1	0	0
LCOM					1	0.01
CBO						1
Mozilla	WMC	DIT	RFC	NOC	LCOM	CBO
WMC	1	0.16	**0.53**	0	**0.64**	**0.39**
DIT		1	**0.54**	0	0.08	**0.23**
RFC			1	0	**0.31**	**0.51**
NOC				1	0	0
LCOM					1	0.16
CBO						1

The bold numbers denote significant correlations.

Table 6.3: Correlations between the metrics of the reference project and Mozilla.

6.5 Studying the changes in Mozilla's metrics

Basili *et al.* drew up six hypotheses (one for each metric) that represent the expected connection between the metrics and the fault-proneness of the code [5]. They tested these hypotheses and found that some of the metrics were very good predictors, while others were not.

Since we wanted to examine how Mozilla's predicted fault-proneness had changed in the last one and a half years, we first calculated these metrics for each version of Mozilla from 1.0 to 1.6. The results can be seen in Table 6.4. Now we are able to use the conclusions drawn in [5] to characterize the changes in Mozilla. We present all the hypotheses and conclusions about the "goodness" of the metrics for detecting fault-proneness as stated in [5] and examine the changes in Mozilla based on their conclusions.

WMC hypothesis: *"A class with significantly more member functions than its peers is more complex and, by consequence, tends to be more fault-prone."* The WMC was found to be somewhat significant in [5]. In Mozilla the rate of classes with a large WMC value decreased slightly but not significantly. We can only say that Mozilla did not get worse according to this metric.

DIT hypothesis: *"A class located deeper in a class inheritance lattice is supposed to be more fault-prone because the class inherits a large number of definitions from its ancestors."* The DIT was found to be very significant in [5], which means that the larger the DIT, the larger the probability of fault-proneness. In Mozilla the rate of classes with seven or more ancestors increased slightly, but the number of these classes is tiny compared to the classes as a whole. On the other hand, the rate of classes which have no ancestors or only one or two increased significantly, while the rate of classes with more than two but fewer than seven ancestors decreased markedly. This suggests that in more recent versions of Mozilla there might be fewer faults.

RFC hypothesis: *"Classes with larger response sets implement more complex functionalities and are, therefore, more fault-prone."* The RFC was shown to be very significant in [5]. The larger the RFC, the larger the probability of fault-proneness. In Mozilla the rate of classes whose RFC value is larger than ten decreased (more than 70% of the classes fall into this group), while the rate of the rest of the classes increased. Overall, this suggests an improvement in quality (so it is less fault-prone).

NOC hypothesis: *"We expect classes with large number of children to be more fault-prone."* The NOC appeared to be very significant but the observed trend is contrary to what was stated by the hypothesis. The larger the NOC the lower the probability of fault-proneness [5]. In Mozilla the number of classes with three or more children is negligible and it did not change significantly. Hence, we only examined the remaining classes. The rate of classes with no or two children decreased while the rate of classes with one child increased. According to this, Mozilla slightly improved in this respect.

WMC	0	1	2	3	4	5	6	7	8	9	10	11-20	21-30	31-40	41-50	50-
1.0	0.75	5.28	11.68	7.95	7.15	6.39	5.51	5.66	5.03	4.97	3.23	18.81	6.71	3.61	2.08	5.20
1.1	0.77	5.31	11.51	7.78	7.22	6.47	5.62	5.64	5.16	4.93	3.32	18.74	6.66	3.63	2.11	5.14
1.2	0.75	4.57	9.67	7.75	7.68	6.12	5.89	5.95	5.16	5.29	3.59	19.23	7.00	3.76	2.09	5.51
1.3	0.85	4.63	9.81	8.10	7.51	6.07	6.03	5.90	5.14	5.22	3.45	19.20	7.00	3.72	2.05	5.35
1.4	0.85	5.05	10.61	8.25	7.39	6.00	5.92	5.92	5.29	4.99	3.50	18.90	6.78	3.48	1.99	5.05
1.5	0.88	5.15	10.49	8.31	7.43	5.97	5.99	5.89	5.35	4.89	3.50	18.97	6.73	3.52	1.86	5.07
1.6	0.88	5.11	10.54	8.56	7.43	5.75	6.03	5.89	5.31	4.91	3.47	18.95	6.63	3.53	1.92	5.09

DIT	0	1	2	3	4	5	6	7	8	9	10	11-20	21-30	31-40	41-50	50-
1.0	15.87	30.21	18.70	11.22	8.41	4.61	4.49	1.34	1.28	0.65	0.92	2.18	0.13	0.00	0.00	0.00
1.1	16.03	30.27	18.66	11.07	8.29	4.67	4.46	1.41	1.33	0.60	0.93	2.16	0.12	0.00	0.00	0.00
1.2	16.82	32.18	19.40	10.76	7.15	3.59	2.86	1.94	1.34	0.64	0.94	2.26	0.13	0.00	0.00	0.00
1.3	16.77	32.43	19.56	10.57	7.21	3.59	2.62	2.01	1.31	0.63	0.97	2.16	0.15	0.02	0.00	0.00
1.4	17.13	32.05	20.56	10.51	6.54	3.00	2.23	2.19	1.79	0.77	0.91	2.05	0.24	0.02	0.00	0.00
1.5	17.30	32.14	20.61	10.51	6.45	2.96	2.16	1.80	1.18	0.96	1.44	2.26	0.24	0.02	0.00	0.00
1.6	17.35	32.32	20.46	10.62	6.15	2.89	2.24	1.86	1.18	0.92	1.44	2.32	0.22	0.02	0.00	0.00

RFC	0	1	2	3	4	5	6	7	8	9	10	11-20	21-30	31-40	41-50	50-
1.0	0.69	1.53	1.19	0.99	2.18	4.88	4.53	3.02	2.98	2.16	2.58	19.87	15.77	7.74	5.70	24.19
1.1	0.70	1.58	1.20	0.89	2.26	4.96	4.42	3.11	3.11	2.20	2.55	19.72	15.82	7.59	5.66	24.24
1.2	0.68	1.62	1.24	0.96	2.43	5.25	4.69	3.37	3.41	2.26	2.84	18.84	13.70	7.49	6.06	25.14
1.3	0.70	1.61	1.23	0.99	2.39	5.64	4.69	3.28	3.49	2.33	2.75	18.52	13.55	7.74	5.98	25.12
1.4	0.66	1.65	1.43	1.15	2.36	5.66	4.83	3.36	3.52	2.48	2.64	19.23	13.69	7.63	5.98	23.74
1.5	0.66	1.68	1.42	1.22	2.38	5.55	4.89	3.36	3.63	2.40	2.76	19.79	13.82	7.31	5.59	23.55
1.6	0.66	1.72	1.48	1.22	2.28	5.65	4.89	3.39	3.43	2.54	2.79	19.82	13.84	7.23	5.43	23.62

NOC	0	1	2	3	4	5	6	7	8	9	10	11-20	21-30	31-40	41-50	50-
1.0	63.19	24.40	6.02	2.33	1.01	0.65	0.55	0.38	0.17	0.10	0.19	0.63	0.19	0.04	0.04	0.13
1.1	63.03	24.49	6.03	2.36	1.04	0.66	0.50	0.35	0.21	0.15	0.17	0.66	0.15	0.06	0.02	0.12
1.2	62.27	25.63	5.68	2.50	0.94	0.73	0.38	0.38	0.19	0.17	0.11	0.70	0.15	0.06	0.00	0.11
1.3	62.05	25.86	5.69	2.45	0.95	0.80	0.40	0.32	0.15	0.19	0.11	0.72	0.15	0.04	0.02	0.11
1.4	62.63	25.81	5.21	2.38	0.87	0.95	0.38	0.34	0.10	0.20	0.10	0.70	0.14	0.06	0.02	0.10
1.5	62.63	25.80	5.23	2.34	0.92	0.96	0.42	0.30	0.12	0.16	0.12	0.70	0.12	0.06	0.02	0.10
1.6	62.67	25.81	5.19	2.34	0.98	0.94	0.38	0.34	0.14	0.16	0.06	0.68	0.12	0.08	0.00	0.10

LCOM	0	1	2	3	4	5	6	7	8	9	10	11-20	21-30	31-40	41-50	50-
1.0	17.40	10.04	0.38	6.86	1.51	0.25	5.70	0.52	0.61	1.47	4.49	7.63	7.74	3.86	2.87	28.68
1.1	17.42	9.91	0.39	6.84	1.60	0.23	5.68	0.56	0.62	1.51	4.46	7.67	7.84	3.92	2.90	28.45
1.2	16.30	8.28	0.38	6.81	1.24	0.21	6.42	0.49	0.66	1.54	4.37	7.87	8.28	3.99	3.24	29.90
1.3	16.17	8.50	0.21	6.87	0.91	0.17	6.26	0.44	0.93	0.66	4.14	9.03	8.22	4.14	3.21	30.13
1.4	16.87	8.94	0.18	7.15	0.99	0.14	6.02	0.36	0.95	0.66	4.11	9.12	8.23	4.15	3.06	29.07
1.5	16.98	8.77	0.20	7.23	0.96	0.18	6.03	0.44	1.04	0.66	4.09	9.07	8.29	3.95	3.08	29.04
1.6	16.99	8.82	0.18	7.35	0.94	0.18	6.19	0.42	1.08	0.62	3.87	9.14	8.19	3.93	3.05	29.05

CBO	0	1	2	3	4	5	6	7	8	9	10	11-20	21-30	31-40	41-50	50-
1.0	42.01	5.95	4.49	3.40	4.72	5.51	6.71	4.47	3.31	2.94	2.68	9.77	2.41	0.99	0.40	0.25
1.1	42.07	5.70	4.67	3.34	4.71	5.72	6.49	4.37	3.30	3.09	2.63	9.79	2.41	1.04	0.44	0.25
1.2	40.52	6.15	4.67	3.69	3.56	6.02	6.87	4.37	3.39	3.14	2.77	10.50	2.65	1.00	0.41	0.28
1.3	40.38	6.45	4.90	3.70	3.26	3.68	5.60	5.79	3.95	3.21	3.21	11.52	2.54	1.10	0.42	0.27
1.4	40.14	7.31	4.63	3.76	3.22	3.72	5.76	5.86	3.78	3.34	2.92	11.54	2.46	0.91	0.40	0.24
1.5	40.24	7.27	4.99	3.69	3.08	3.69	5.75	5.85	3.56	3.40	2.84	11.68	2.36	1.02	0.34	0.24
1.6	40.29	7.47	4.95	3.75	3.15	3.55	5.65	5.77	3.67	3.23	2.93	11.58	2.32	1.08	0.38	0.24

The numbers represent the percentage of the number of classes having the metric value in its column header.

Table 6.4: Metrics of the seven versions of Mozilla.

LCOM hypothesis: *"Classes with low cohesion among its methods suggests an inappropriate design which is likely to be more fault-prone."* The LCOM was stated to be insignificant in [5], but according to the hypothesis Mozilla got slightly worse because the rate of classes with the value eleven or more increased slightly while the rest remained about the same.

CBO hypothesis: *"Highly coupled classes are more fault-prone than weakly coupled classes."* The CBO was said to be significant in [5] but it is hard to say anything about Mozilla using this metric. The rate of classes whose CBO value is one, two or three increased and the rate of classes whose CBO value is four, five or six decreased, which is good and may suggest an increase in quality. On the other hand, the rate of classes with large CBO values increased, which suggests more faults.

So far we have compared the distribution of the metrics collected from seven different versions of Mozilla. This is very useful if we are interested in learning how the whole system changed but it means little if we want to examine only some classes of interest. In that case, we have to study the changes in the metrics for the given class. We may illustrate this using one of the biggest classes. We chose *nsHTMLEditor* because three out of the six of its metrics are maximal in the whole system (see Table 6.5). As we can see, more than a hundred new methods have been added (WMC) and its response set grew by more than two hundred (RFC). The DIT also grew by 30%. But LCOM changed the most significantly because it is twice as big as it was in version 1.0. According to these values, this class became much more fault-prone.

ver.	WMC	DIT	RFC	NOC	LCOM	CBO
1.0	235	13	843	0	27,233	50
1.1	238	13	852	0	27,939	50
1.2	250	13	885	0	30,805	51
1.3	256	14	902	0	32,262	51
1.4	295	15	1,005	0	42,529	57
1.5	337	17	1,071	0	55,200	56
1.6	337	17	1,074	0	55,198	57

Table 6.5: Metrics of the class *nsHTMLEditor*.

6.6 Summary

This chapter makes three key contributions: (1) we showed how our compiler wrapper toolset realizes the fact extraction process in practice on real-world software; (2) using the collected facts, we calculated object-oriented metrics and supplemented a previous work [5] with measurements made on the real-world software package called Mozilla; and (3) using the calculated metrics we studied how Mozilla's predicted fault-proneness had changed over seven versions covering one and a half years of development.

When checking the seven versions of Mozilla we found that the predicted fault-proneness of the software as a whole decreased slightly, but we also found example classes with an increased probability of fault-proneness. This suggests that the Mozilla development community should concentrate even more on improving the quality of the source code.

"You can't get where you want to go
if you don't know where you are."
Anonymous

Chapter 7

Conclusions

With this work we achieved it is now possible to reverse engineer the source code of large, real-world software systems written in the C++ programming language. What is more, we can do this without having to modify any source code, not even the makefiles or project files.

During the fact extraction process our framework prepares a model of the analyzed C++ system according to a well-defined schema. Having such a schema enables the community and industry to improve existing reengineering tools and develop new ones that can seamlessly exchange information about the software system being studied. Our framework contains an application programming interface based on this schema with which the extracted information can be easily accessed and used. Output files in various formats can also be produced to promote tool interoperability.

Containing various algorithms for producing derived outputs – like object-oriented metrics – as well, the framework provides a complete solution for reverse engineering C++ code so that it relieves researchers of the burden of having to write parsers for various purposes and allows them to focus on their own concrete research topic.

7.1 Fulfillment of the requirements

We will now evaluate the Columbus framework and schema for C++ according to the requirements given in Section 2.5.

1. **Analysis of source files**. To support the analysis of source files we developed two command line programs, which are both robust and fault-tolerant. That is, they have the ability to parse incomplete, syntactically incorrect source code as well (see Section 4.2.4).

    - *Extracting preprocessor-related facts*. The Columbus framework contains a tool called CANPP (C/C++ ANalyzer-PreProcessor) which is able to analyze C/C++ preprocessor-related language constructs, to build up a preprocessing schema instance and to preprocess the code.

- *Extracting C++ language-related facts*. In order to analyze the preprocessed C++ source code, our framework contains a tool called CAN (C++ ANalyzer). During the analysis it builds up the schema instance of the compilation unit being analyzed. Apart from standard C++, CAN can handle the Microsoft, Borland and GCC dialects as well.

2. **Capability for handling large projects**. The Columbus framework has powerful project handling features. One of the main requirements of the framework in the design phase was to include support for handling real-world software systems.
   - *Acquiring project/configuration information*. To make use of the information found in makefiles the framework contains a so-called compiler wrapper toolset which handles the project/configuration automatically (see Sections 4.2.3 and 6.2). To handle IDE project files two different approaches are available: IDE integration, where the framework uses the project handling capabilities of the IDE itself (see Section 4.2.2) and project file import, where Columbus parses existing project files and extracts information from them (see Section 4.2.1).
   - *Filtering*. Columbus offers powerful filtering methods. The default filtering already removes those C++ elements that come from the standard libraries and leaves only the user's own classes in the filter (see Section 4.2.5).
   - *Performance*. We can say that the analyzers are rather fast. They analyzed the open source web and e-mail suite Mozilla [50; 56] in less than three hours with the help of our compiler wrapper toolset.
3. **Connectivity with other tools**. The Columbus framework itself is not a full-feature reengineering environment. It takes care only of the reverse engineering and filtering part of a full reengineering process. For instance, Columbus does not have real support for visualization (just a simple tree-view in the filter window that lists the classes and namespaces in their scoping structure). Instead it passes this job over to other tools that have been integrated with Columbus (like, for example, rigi [51] and CodeCrawler [19]).
   - *Fact representation*. The Columbus Schema for C++ represents the syntax and the semantics of the subject system in great detail. However, some higher level information such as data dependencies are not provided. The schema is modular: it is divided into six packages, and the connection between them is rather weak. This means that it is easily extendible (e.g. the type representation package can easily be replaced if required). An instance of the schema is a graph, so it can be easily represented in any desired physical form. Some of these like GXL [39] and RSF [51] have already been implemented. The schema contains an attribute for storing non-standard C++ specifiers and, apart from this, the modular structure of the schema allows its easy extension to support arbitrary language dialects and extensions (see Section 3.1).

- *Schema instance conversions*. Columbus is highly extendible due to its plug-in architecture. The framework already contains several conversion algorithms, so the extracted facts can be easily accessed with other tools (see Section 4.2.6). In addition, the user can easily write and add his/her own new plug-in to the Columbus system using an easy-to-use plug-in API (see Section 4.2.1).

7.2 Utilization of the new results

We have successfully utilized our fact extraction and representation technology to achieve two further goals. First, we developed new methods for recognizing design pattern occurrences in C++ source code and second, we analyzed several versions of the open-source internet suite Mozilla to study the changes in its predicted fault-proneness and quality.

We developed two methods for recognizing design patterns in C++ source code. First, we integrated our Columbus framework with the metrics calculator and design pattern miner tool Maisa. We analyzed the C++ code with Columbus and the collected facts were converted to a form understandable to Maisa. Afterwards, this file was analyzed by Maisa, and the reference implementations of various design patterns were identified. Second, we developed a new approach to the problem of pattern mining which includes the detection of call delegations, object creations and operation redefinitions. We also designed a new file format for describing design patterns to be searched for. This is an XML-based format called the Design Pattern Markup Language (DPML). It gives the user the freedom to modify the pattern descriptions and also to create new ones. We tested our method on four large public-domain projects and achieved valuable results.

We analyzed Mozilla and obtained three results. First, we showed that our framework has the capabilities to reverse engineer the source code of a real-world software system by simply typing two command lines. Then, we supplemented a valuable previous work [5] with measurements made on a large software system with more than a million lines of code. Finally, using the calculated metrics, we studied how Mozilla's predicted fault-proneness had changed over seven versions covering over one and a half years of development.

Appendix A

CPPML – C++ Markup Language

```
<!-- Global Entities  -->
<!ENTITY % Boolean "(true|false)">
<!ENTITY % ProjectAttr
  "name CDATA #REQUIRED
   path CDATA #IMPLIED
   xmlns:base  CDATA #FIXED 'columbus_cpp_schema/base'
   xmlns:struc CDATA #FIXED 'columbus_cpp_schema/struc'
   xmlns:type  CDATA #FIXED 'columbus_cpp_schema/type'
   xmlns:templ CDATA #FIXED 'columbus_cpp_schema/templ'
   xmlns:statm CDATA #FIXED 'columbus_cpp_schema/statm'
   xmlns:expr  CDATA #FIXED 'columbus_cpp_schema/expr'">

<!-- base Entities  -->
<!ENTITY % base_AccessibilityKind "(ackNone|ackPrivate|ackProtected|ackPublic)">
<!ENTITY % base_StorageClassKind "(sckNone|sckAuto|sckRegister|sckStatic|sckExtern|sckMutable)">
<!ENTITY % base_ClassKind "(clkClass|clkStruct|clkUnion)">
<!ENTITY % base_UsingKind "(uskDirective|uskDeclaration)">
<!ENTITY % base_FunctionKind "(fnkNormal|fnkConstructor|fnkDestructor|fnkOperator|fnkConversion)">
<!ENTITY % base_ConstVolatileKind "(cvkNone|cvkConst|cvkVolatile|cvkConstVolatile)">
<!ENTITY % base_PointerKind "(ptkPointer|ptkReference|ptkPointerToMember)">
<!ENTITY % base_SimpleTypeKind "(stkChar|stkUnsignedChar|stkSignedChar|stkBool|stkUnsignedInt|stkInt
                               |stkUnsignedShortInt|stkUnsignedLongInt|stkLongInt|stkShortInt
                               |stkWchar_t|stkFloat|stkDouble|stkLongDouble|stkVoid|stkNonISO)">
<!ENTITY % base_NamedConversionKind "(nckDynamic|nckStatic|nckReinterpret|nckConst)">
<!ENTITY % base_IncDecKind "(idkInc|idkDec)">
<!ENTITY % base_MemberSelectionKind "(mskDot|mskArrow)">
<!ENTITY % base_PointerToMemberKind "(pmkDotPointer|pmkArrowPointer)">
<!ENTITY % base_AssignmentKind "(askAssignEqual|askTimesEqual|askDivideEqual|askModEqual
                               |askPlusEqual|askMinusEqual|askShiftRightEqual|askShiftLeftEqual
                               |askBitwiseAndEqual|askBitwiseXorEqual|askBitwiseOrEqual)">
<!ENTITY % base_UnaryArithmeticKind "(uakPlus|uakMinus|uakComplement)">
<!ENTITY % base_BinaryArithmeticKind "(bakStar|bakDivide|bakMod|bakPlus|bakMinus|bakShiftLeft
                                     |bakShiftRight|bakBitwiseAnd|bakBitwiseXor|bakBitwiseOr)">
<!ENTITY % base_BinaryLogicalKind "(blkLessThan|blkGreaterThan|blkLessThanOrEqualTo
                                  |blkGreaterThanOrEqualTo|blkEqual|blkNotEqual|blkAnd|blkOr)">
<!ENTITY % base_BaseAttr
  "id ID #REQUIRED">

<!ENTITY % base_PositionedAttr
```

```
  "path    CDATA #IMPLIED
   line    CDATA #IMPLIED
   col     CDATA #IMPLIED
   endLine CDATA #IMPLIED
   endCol  CDATA #IMPLIED
   comment CDATA #IMPLIED">
<!ENTITY % base_NamedAttr "name CDATA #IMPLIED">

<!-- struc Entities  -->
<!ENTITY % struc_MemberAttr
  "accessibility %base_AccessibilityKind; #IMPLIED
   storageClass  %base_StorageClassKind;  #IMPLIED
   linkageSpecification    CDATA          #IMPLIED
   nonISOSpec    CDATA                    #IMPLIED">
<!ENTITY % struc_ScopeAttr "">
<!ENTITY % struc_ClassAttr
  "kind       %base_ClassKind; #IMPLIED
   isAbstract %Boolean;        #IMPLIED
   isDefined  %Boolean;        #IMPLIED">
<!ENTITY % struc_ClassTemplAttr "">
<!ENTITY % struc_ClassTemplSpecAttr "">
<!ENTITY % struc_BaseSpecifierAttr
  "accessibility %base_AccessibilityKind; #IMPLIED
   isVirtual     %Boolean;                #IMPLIED">
<!ENTITY % struc_FriendSpecifierAttr "">
<!ENTITY % struc_NamespaceAttr "">
<!ENTITY % struc_NamespaceAliasAttr "">
<!ENTITY % struc_UsingAttr "kind %base_UsingKind; #REQUIRED">
<!ENTITY % struc_TemplInstanceAttr "">
<!ENTITY % struc_EnumerationAttr "isDefined %Boolean; #IMPLIED">
<!ENTITY % struc_EnumeratorAttr "">
<!ENTITY % struc_FunctionAttr
  "mangledName   CDATA               #IMPLIED
   kind          %base_FunctionKind; #IMPLIED
   isVirtual     %Boolean;           #IMPLIED
   isPureVirtual %Boolean;           #IMPLIED
   isInline      %Boolean;           #IMPLIED
   isExplicit    %Boolean;           #IMPLIED">
<!ENTITY % struc_FunctionTemplAttr "">
<!ENTITY % struc_FunctionTemplSpecAttr "">
<!ENTITY % struc_ObjectAttr "">
<!ENTITY % struc_TypedefAttr "">
<!ENTITY % struc_AsmAttr "text CDATA #IMPLIED">
<!ENTITY % struc_ParameterAttr "isEllipsis %Boolean; #REQUIRED">
<!ENTITY % struc_MemInitializerAttr "">

<!-- type Entities  -->
<!ENTITY % type_SimpleTypeAttr "kind %base_SimpleTypeKind; #IMPLIED">
<!ENTITY % type_TypeFormerAttr "">
<!ENTITY % type_TypeRepAttr "">
<!ENTITY % type_TypeFormerTypeAttr "constVolatile %base_ConstVolatileKind; #IMPLIED">
<!ENTITY % type_TypeFormerArrAttr "">
<!ENTITY % type_TypeFormerPtrAttr
  "kind          %base_PointerKind;       #IMPLIED
   constVolatile %base_ConstVolatileKind; #IMPLIED">
<!ENTITY % type_TypeFormerFuncAttr "constVolatile %base_ConstVolatileKind; #IMPLIED">
```

```
<!-- templ Entities  -->
<!ENTITY % templ_ParameterAttr "">
<!ENTITY % templ_ParameterTypeAttr "">
<!ENTITY % templ_ParameterNonTypeAttr "">
<!ENTITY % templ_ParameterTemplAttr "">
<!ENTITY % templ_ParameterListAttr "">
<!ENTITY % templ_ArgumentListAttr "">
<!ENTITY % templ_ArgumentAttr "">
<!ENTITY % templ_ArgumentTemplAttr "">
<!ENTITY % templ_ArgumentNonTypeAttr "">
<!ENTITY % templ_ArgumentTypeAttr "">

<!-- statm Entities  -->
<!ENTITY % statm_StatementAttr "">
<!ENTITY % statm_BlockAttr "">
<!ENTITY % statm_TryBlockAttr "">
<!ENTITY % statm_HandlerAttr "">
<!ENTITY % statm_CatchParameterAttr "isEllipsis %Boolean; #REQUIRED">
<!ENTITY % statm_SelectionAttr "">
<!ENTITY % statm_IfAttr "">
<!ENTITY % statm_SwitchAttr "">
<!ENTITY % statm_IterationAttr "">
<!ENTITY % statm_WhileAttr "">
<!ENTITY % statm_ForAttr "">
<!ENTITY % statm_DoAttr "">
<!ENTITY % statm_JumpAttr "">
<!ENTITY % statm_BreakAttr "">
<!ENTITY % statm_ContinueAttr "">
<!ENTITY % statm_ReturnAttr "">
<!ENTITY % statm_GotoAttr "">
<!ENTITY % statm_EmptyAttr "">
<!ENTITY % statm_LabelAttr "">
<!ENTITY % statm_CaseLabelAttr "">
<!ENTITY % statm_DefaultLabelAttr "">
<!ENTITY % statm_IdLabelAttr "name CDATA #IMPLIED">

<!-- expr Entities  -->
<!ENTITY % expr_ExpressionAttr "">
<!ENTITY % expr_ExpressionListAttr "">
<!ENTITY % expr_NewAttr "isGlobal %Boolean; #REQUIRED">
<!ENTITY % expr_IdAttr "name CDATA #IMPLIED">
<!ENTITY % expr_TypeidTypeAttr "">
<!ENTITY % expr_ConditionalAttr "">
<!ENTITY % expr_SizeofTypeAttr "">
<!ENTITY % expr_ThisAttr "">
<!ENTITY % expr_ThrowAttr "">
<!ENTITY % expr_FunctionalConversionAttr "">
<!ENTITY % expr_UnaryAttr "">
<!ENTITY % expr_FunctionCallAttr "">
<!ENTITY % expr_NamedConversionAttr "kind %base_NamedConversionKind; #REQUIRED">
<!ENTITY % expr_CastConversionAttr "">
<!ENTITY % expr_PostIncDecAttr "kind %base_IncDecKind; #REQUIRED">
<!ENTITY % expr_PreIncDecAttr "kind %base_IncDecKind; #REQUIRED">
<!ENTITY % expr_TypeidExprAttr "">
<!ENTITY % expr_IndirectionAttr "">
```

```
<!ENTITY % expr_AddressOfAttr "">
<!ENTITY % expr_UnaryArithmeticAttr "kind %base_UnaryArithmeticKind; #REQUIRED">
<!ENTITY % expr_UnaryLogicalAttr "">
<!ENTITY % expr_SizeofExprAttr "">
<!ENTITY % expr_DeleteAttr
  "isGlobal %Boolean; #REQUIRED
   isArray  %Boolean; #REQUIRED">
<!ENTITY % expr_BinaryAttr "">
<!ENTITY % expr_AssignmentAttr "kind %base_AssignmentKind; #REQUIRED">
<!ENTITY % expr_ArraySubscriptAttr "">
<!ENTITY % expr_MemberSelectionAttr "kind %base_MemberSelectionKind; #REQUIRED">
<!ENTITY % expr_CommaAttr "">
<!ENTITY % expr_PointerToMemberAttr "kind %base_PointerToMemberKind; #REQUIRED">
<!ENTITY % expr_BinaryArithmeticAttr "kind %base_BinaryArithmeticKind; #REQUIRED">
<!ENTITY % expr_BinaryLogicalAttr "kind %base_BinaryLogicalKind; #REQUIRED">
<!ENTITY % expr_LiteralAttr "">
<!ENTITY % expr_IntegerLiteralAttr "value CDATA #IMPLIED">
<!ENTITY % expr_CharacterLiteralAttr "value CDATA #IMPLIED">
<!ENTITY % expr_FloatingLiteralAttr "value CDATA #IMPLIED">
<!ENTITY % expr_StringLiteralAttr "value CDATA #IMPLIED">
<!ENTITY % expr_BooleanLiteralAttr "value %Boolean; #IMPLIED">

<!-- Helper Entities  -->
<!ENTITY % classMembers "struc:Class|struc:ClassTempl|struc:ClassTemplSpec|struc:Using
                         |struc:TemplInstance|struc:Enumeration|struc:Enumerator|struc:Function
                         |struc:FunctionTempl|struc:FunctionTemplSpec|struc:Object|struc:Typedef">
<!ENTITY % statements "statm:Block|statm:TryBlock|statm:Handler|statm:If|statm:Switch|statm:While
                      |statm:For|statm:Do|statm:Break|statm:Continue|statm:Return|statm:Goto
                      |statm:Empty">
<!ENTITY % expressions "expr:New|expr:Id|expr:TypeidType|expr:Conditional|expr:SizeofType|expr:This
                       |expr:Throw|expr:FunctionalConversion|expr:NamedConversion
                       |expr:CastConversion|expr:PostIncDec|expr:PreIncDec|expr:TypeidExpr
                       |expr:Indirection|expr:AddressOf|expr:UnaryArithmetic|expr:UnaryLogical
                       |expr:SizeofExpr|expr:Delete|expr:Assignment|expr:ArraySubscript
                       |expr:MemberSelection|expr:Comma|expr:PointerToMember|expr:BinaryArithmetic
                       |expr:BinaryLogical|expr:FunctionCall|expr:IntegerLiteral
                       |expr:CharacterLiteral|expr:FloatingLiteral|expr:StringLiteral
                       |expr:BooleanLiteral">

<!-- Main Elements  -->
<!ELEMENT Project (struc:Namespace, type:SimpleType*, type:TypeRep*, Filtered?)>
<!ATTLIST Project %ProjectAttr;>

<!ELEMENT Filtered EMPTY>
<!ATTLIST Filtered %base_BaseAttr;>

<!-- struc Elements  -->
<!ELEMENT struc:Class (struc:declares?, struc:hasBaseSpecifier*,
                       struc:hasFriendSpecifier*, (%classMembers;)*)>
<!ATTLIST struc:Class %base_BaseAttr;
                      %base_PositionedAttr;
                      %base_NamedAttr;
                      %struc_MemberAttr;
                      %struc_ScopeAttr;
                      %struc_ClassAttr;>
<!ELEMENT struc:ClassTempl (struc:declares?, struc:hasTemplParameters,
```

```
                          struc:hasBaseSpecifier*, struc:hasFriendSpecifier*,
                          (%classMembers;)*)>
<!ATTLIST struc:ClassTempl %base_BaseAttr;
                          %base_PositionedAttr;
                          %base_NamedAttr;
                          %struc_MemberAttr;
                          %struc_ScopeAttr;
                          %struc_ClassAttr;
                          %struc_ClassTemplAttr;>
<!ELEMENT struc:ClassTemplSpec (struc:declares?, struc:specializes,
                            struc:hasTemplParameters, struc:hasTemplArguments,
                            struc:hasBaseSpecifier*,
                            struc:hasFriendSpecifier*, (%classMembers;)*)>
<!ATTLIST struc:ClassTemplSpec %base_BaseAttr;
                            %base_PositionedAttr;
                            %base_NamedAttr;
                            %struc_MemberAttr;
                            %struc_ScopeAttr;
                            %struc_ClassAttr;
                            %struc_ClassTemplAttr;
                            %struc_ClassTemplSpecAttr;>
<!ELEMENT struc:BaseSpecifier (struc:derivesFrom, struc:hasTemplArguments*)>
<!ATTLIST struc:BaseSpecifier %base_BaseAttr;
                           %base_PositionedAttr;
                           %struc_BaseSpecifierAttr;>
<!ELEMENT struc:FriendSpecifier (struc:grantsFriendship, struc:hasTemplArguments*)>
<!ATTLIST struc:FriendSpecifier %base_BaseAttr;
                             %base_PositionedAttr;
                             %struc_FriendSpecifierAttr;>
<!ELEMENT struc:Namespace (struc:Namespace|struc:NamespaceAlias|struc:Asm|%classMembers;)*>
<!ATTLIST struc:Namespace %base_BaseAttr;
                         %base_PositionedAttr;
                         %base_NamedAttr;
                         %struc_MemberAttr;
                         %struc_ScopeAttr;
                         %struc_NamespaceAttr;>
<!ELEMENT struc:NamespaceAlias (struc:refersToNamespace)>
<!ATTLIST struc:NamespaceAlias %base_BaseAttr;
                            %base_PositionedAttr;
                            %base_NamedAttr;
                            %struc_MemberAttr;
                            %struc_NamespaceAliasAttr;>
<!ELEMENT struc:Using (struc:refersToMember)>
<!ATTLIST struc:Using %base_BaseAttr;
                     %base_PositionedAttr;
                     %base_NamedAttr;
                     %struc_MemberAttr;
                     %struc_UsingAttr;>
<!ELEMENT struc:TemplInstance (struc:instantiates, struc:hasTemplArguments)>
<!ATTLIST struc:TemplInstance %base_BaseAttr;
                           %base_PositionedAttr;
                           %base_NamedAttr;
                           %struc_MemberAttr;
                           %struc_TemplInstanceAttr;>
<!ELEMENT struc:Enumeration (struc:declares?, struc:hasEnumerator*)>
<!ATTLIST struc:Enumeration %base_BaseAttr;
```

```
                          %base_PositionedAttr;
                          %base_NamedAttr;
                          %struc_MemberAttr;
                          %struc_EnumerationAttr;>
<!ELEMENT struc:Enumerator (struc:isEnumeratorOf, struc:hasValue?)>
<!ATTLIST struc:Enumerator %base_BaseAttr;
                           %base_PositionedAttr;
                           %base_NamedAttr;
                           %struc_MemberAttr;
                           %struc_EnumeratorAttr;>
<!ELEMENT struc:Function (struc:declares?, struc:hasTypeRep, struc:Parameter*, struc:throwsTypeRep*,
                          struc:hasIdLabel*, struc:hasConstructorInitializer*, struc:hasBody?)>
<!ATTLIST struc:Function %base_BaseAttr;
                         %base_PositionedAttr;
                         %base_NamedAttr;
                         %struc_MemberAttr;
                         %struc_FunctionAttr;>
<!ELEMENT struc:FunctionTempl (struc:declares?, struc:hasTemplParameters, struc:hasTypeRep,
                               struc:Parameter*, struc:throwsTypeRep*, struc:hasIdLabel*,
                               struc:hasConstructorInitializer*, struc:hasBody?)>
<!ATTLIST struc:FunctionTempl %base_BaseAttr;
                              %base_PositionedAttr;
                              %base_NamedAttr;
                              %struc_MemberAttr;
                              %struc_FunctionAttr;
                              %struc_FunctionTemplAttr;>
<!ELEMENT struc:FunctionTemplSpec (struc:declares?, struc:specializes, struc:hasTemplParameters,
                                   struc:hasTemplArguments, struc:hasTypeRep, struc:Parameter*,
                                   struc:throwsTypeRep*, struc:hasIdLabel*,
                                   struc:hasConstructorInitializer*, struc:hasBody?)>
<!ATTLIST struc:FunctionTemplSpec %base_BaseAttr;
                                  %base_PositionedAttr;
                                  %base_NamedAttr;
                                  %struc_MemberAttr;
                                  %struc_FunctionAttr;
                                  %struc_FunctionTemplAttr;
                                  %struc_FunctionTemplSpecAttr;>
<!ELEMENT struc:Object (struc:declares?, struc:hasTypeRep, struc:hasInitValue?, struc:hasBitfield?)>
<!ATTLIST struc:Object %base_BaseAttr;
                       %base_PositionedAttr;
                       %base_NamedAttr;
                       %struc_MemberAttr;
                       %struc_ObjectAttr;>
<!ELEMENT struc:Typedef (struc:hasTypeRep)>
<!ATTLIST struc:Typedef %base_BaseAttr;
                        %base_PositionedAttr;
                        %base_NamedAttr;
                        %struc_MemberAttr;
                        %struc_TypedefAttr;>
<!ELEMENT struc:Asm EMPTY>
<!ATTLIST struc:Asm %base_BaseAttr;
                    %base_PositionedAttr;
                    %base_NamedAttr;
                    %struc_MemberAttr;
                    %struc_AsmAttr;>
<!ELEMENT struc:Parameter (struc:hasTypeRep, struc:hasDefValue?)>
```

```
<!ATTLIST struc:Parameter %base_BaseAttr;
                          %base_PositionedAttr;
                          %base_NamedAttr;
                          %struc_ParameterAttr;>

<!ELEMENT struc:MemInitializer (struc:initializes, struc:hasArguments)>
<!ATTLIST struc:MemInitializer %base_BaseAttr;
                               %base_PositionedAttr;
                               %struc_MemInitializerAttr;>
<!ELEMENT struc:declares EMPTY>
<!ATTLIST struc:declares ref IDREF #REQUIRED>

<!ELEMENT struc:hasBaseSpecifier (struc:BaseSpecifier)>
<!ATTLIST struc:hasBaseSpecifier>

<!ELEMENT struc:hasFriendSpecifier (struc:FriendSpecifier)>
<!ATTLIST struc:hasFriendSpecifier>

<!ELEMENT struc:hasTemplParameters (templ:ParameterList)>
<!ATTLIST struc:hasTemplParameters>

<!ELEMENT struc:hasTemplArguments (templ:ArgumentList)>
<!ATTLIST struc:hasTemplArguments>

<!ELEMENT struc:specializes EMPTY>
<!ATTLIST struc:specializes ref IDREF #REQUIRED>

<!ELEMENT struc:derivesFrom EMPTY>
<!ATTLIST struc:derivesFrom ref IDREF #REQUIRED>

<!ELEMENT struc:grantsFriendship EMPTY>
<!ATTLIST struc:grantsFriendship ref IDREF #REQUIRED>

<!ELEMENT struc:refersToNamespace EMPTY>
<!ATTLIST struc:refersToNamespace ref IDREF #REQUIRED>

<!ELEMENT struc:refersToMember EMPTY>
<!ATTLIST struc:refersToMember ref IDREF #REQUIRED>

<!ELEMENT struc:instantiates EMPTY>
<!ATTLIST struc:instantiates ref IDREF #REQUIRED>

<!ELEMENT struc:hasEnumerator EMPTY>
<!ATTLIST struc:hasEnumerator ref IDREF #REQUIRED>

<!ELEMENT struc:isEnumeratorOf EMPTY>
<!ATTLIST struc:isEnumeratorOf ref IDREF #REQUIRED>

<!ELEMENT struc:hasValue (%expressions;)>
<!ATTLIST struc:hasValue>

<!ELEMENT struc:throwsTypeRep EMPTY>
<!ATTLIST struc:throwsTypeRep ref IDREF #REQUIRED>

<!ELEMENT struc:hasIdLabel (statm:IdLabel)>
<!ATTLIST struc:hasIdLabel>
```

```
<!ELEMENT struc:hasConstructorInitializer (struc:MemInitializer)>
<!ATTLIST struc:hasConstructorInitializer>

<!ELEMENT struc:hasBody (statm:Block|statm:TryBlock)>
<!ATTLIST struc:hasBody>

<!ELEMENT struc:hasInitValue (%expressions;|expr:ExpressionList)>
<!ATTLIST struc:hasInitValue>

<!ELEMENT struc:hasBitfield (%expressions;)>
<!ATTLIST struc:hasBitfield>

<!ELEMENT struc:hasDefValue (%expressions;)>
<!ATTLIST struc:hasDefValue>

<!ELEMENT struc:initializes EMPTY>
<!ATTLIST struc:initializes ref IDREF #REQUIRED>

<!ELEMENT struc:hasTypeRep EMPTY>
<!ATTLIST struc:hasTypeRep ref IDREF #REQUIRED>

<!ELEMENT struc:hasArguments (expr:ExpressionList)>
<!ATTLIST struc:hasArguments>

<!-- type Elements  -->
<!ELEMENT type:TypeRep (type:TypeFormerType|type:TypeFormerArr|
                        type:TypeFormerPtr|type:TypeFormerFunc)*>
<!ATTLIST type:TypeRep %base_BaseAttr;
                       %type_TypeRepAttr;>
<!ELEMENT type:SimpleType EMPTY>
<!ATTLIST type:SimpleType %base_BaseAttr;
                          %type_SimpleTypeAttr;>
<!ELEMENT type:TypeFormerType (type:refersToType)>
<!ATTLIST type:TypeFormerType %base_BaseAttr;
                              %type_TypeFormerAttr;
                              %type_TypeFormerTypeAttr;>
<!ELEMENT type:TypeFormerArr (type:arraySize?)>
<!ATTLIST type:TypeFormerArr %base_BaseAttr;
                             %type_TypeFormerAttr;
                             %type_TypeFormerArrAttr;>
<!ELEMENT type:TypeFormerPtr (type:pointerToMember?)>
<!ATTLIST type:TypeFormerPtr %base_BaseAttr;
                             %type_TypeFormerAttr;
                             %type_TypeFormerPtrAttr;>
<!ELEMENT type:TypeFormerFunc (type:hasReturnTypeRep, type:hasParameterTypeRep*)>
<!ATTLIST type:TypeFormerFunc %base_BaseAttr;
                              %type_TypeFormerAttr;
                              %type_TypeFormerFuncAttr;>
<!ELEMENT type:refersToType EMPTY>
<!ATTLIST type:refersToType ref IDREF #REQUIRED>

<!ELEMENT type:arraySize (%expressions;)>
<!ATTLIST type:arraySize>

<!ELEMENT type:pointerToMember EMPTY>
```

```
<!ATTLIST type:pointerToMember ref IDREF #IMPLIED>

<!ELEMENT type:hasTypeRep EMPTY>
<!ATTLIST type:hasTypeRep ref IDREF #REQUIRED>

<!ELEMENT type:hasReturnTypeRep EMPTY>
<!ATTLIST type:hasReturnTypeRep ref IDREF #REQUIRED>

<!ELEMENT type:hasParameterTypeRep EMPTY>
<!ATTLIST type:hasParameterTypeRep ref IDREF #REQUIRED>

<!-- templ Elements  -->
<!ELEMENT templ:ParameterList (templ:ParameterType|templ:ParameterNonType|templ:ParameterTempl)*>
<!ATTLIST templ:ParameterList %base_BaseAttr;
                              %base_PositionedAttr;
                              %templ_ParameterListAttr;>
<!ELEMENT templ:ParameterType (templ:hasDefTypeRep?)>
<!ATTLIST templ:ParameterType %base_BaseAttr;
                              %base_PositionedAttr;
                              %base_NamedAttr;
                              %templ_ParameterAttr;
                              %templ_ParameterTypeAttr;>
<!ELEMENT templ:ParameterNonType (templ:hasTypeRep, templ:hasDefValue?)>
<!ATTLIST templ:ParameterNonType %base_BaseAttr;
                                 %base_PositionedAttr;
                                 %base_NamedAttr;
                                 %templ_ParameterAttr;
                                 %templ_ParameterNonTypeAttr;>
<!ELEMENT templ:ParameterTempl (templ:hasTemplParameters, templ:refersToDefTempl?)>
<!ATTLIST templ:ParameterTempl %base_BaseAttr;
                               %base_PositionedAttr;
                               %base_NamedAttr;
                               %templ_ParameterAttr;
                               %templ_ParameterTemplAttr;>
<!ELEMENT templ:ArgumentList (templ:ArgumentTempl|templ:ArgumentNonType|templ:ArgumentType)*>
<!ATTLIST templ:ArgumentList %base_BaseAttr;
                             %base_PositionedAttr;
                             %templ_ArgumentListAttr;>
<!ELEMENT templ:ArgumentTempl (templ:refersToTempl)>
<!ATTLIST templ:ArgumentTempl %base_BaseAttr;
                              %base_PositionedAttr;
                              %templ_ArgumentAttr;
                              %templ_ArgumentTemplAttr;>
<!ELEMENT templ:ArgumentNonType (templ:hasValue)>
<!ATTLIST templ:ArgumentNonType %base_BaseAttr;
                                %base_PositionedAttr;
                                %templ_ArgumentAttr;
                                %templ_ArgumentNonTypeAttr;>
<!ELEMENT templ:ArgumentType (templ:hasTypeRep)>
<!ATTLIST templ:ArgumentType %base_BaseAttr;
                             %base_PositionedAttr;
                             %templ_ArgumentAttr;
                             %templ_ArgumentTypeAttr;>
<!ELEMENT templ:hasDefTypeRep EMPTY>
<!ATTLIST templ:hasDefTypeRep ref IDREF #REQUIRED>
```

```
<!ELEMENT templ:hasDefValue (%expressions;)>
<!ATTLIST templ:hasDefValue>

<!ELEMENT templ:hasTemplParameters (templ:ParameterList)>
<!ATTLIST templ:hasTemplParameters>

<!ELEMENT templ:refersToDefTempl EMPTY>
<!ATTLIST templ:refersToDefTempl ref IDREF #REQUIRED>

<!ELEMENT templ:refersToTempl EMPTY>
<!ATTLIST templ:refersToTempl ref IDREF #REQUIRED>

<!ELEMENT templ:hasValue (%expressions;)>
<!ATTLIST templ:hasValue>

<!ELEMENT templ:hasTypeRep EMPTY>
<!ATTLIST templ:hasTypeRep ref IDREF #REQUIRED>

<!-- statm Elements  -->
<!ELEMENT statm:Block (%statements;|%expressions;|%classMembers;|
                       struc:NamespaceAlias|struc:Asm)*>
<!ATTLIST statm:Block %base_BaseAttr;
                      %base_PositionedAttr;
                      %statm_StatementAttr;
                      %statm_BlockAttr;>
<!ELEMENT statm:TryBlock (statm:hasBlock, statm:hasHandler+)>
<!ATTLIST statm:TryBlock %base_BaseAttr;
                         %base_PositionedAttr;
                         %statm_StatementAttr;
                         %statm_TryBlockAttr;>
<!ELEMENT statm:Handler (statm:hasBlock, statm:hasCatchParameter)>
<!ATTLIST statm:Handler %base_BaseAttr;
                        %base_PositionedAttr;
                        %statm_StatementAttr;
                        %statm_HandlerAttr;>
<!ELEMENT statm:CatchParameter (statm:hasTypeRep)>
<!ATTLIST statm:CatchParameter %base_BaseAttr;
                               %base_PositionedAttr;
                               %base_NamedAttr;
                               %statm_CatchParameterAttr;>
<!ELEMENT statm:If (statm:hasCondition, statm:hasSubstatement*, statm:hasFalseSubstatement?)>
<!ATTLIST statm:If %base_BaseAttr;
                   %base_PositionedAttr;
                   %statm_StatementAttr;
                   %statm_SelectionAttr;
                   %statm_IfAttr;>
<!ELEMENT statm:Switch (statm:hasCondition, statm:hasSubstatement*,
                        statm:hasCaseLabel*,statm:hasDefaultLabel?)>
<!ATTLIST statm:Switch %base_BaseAttr;
                       %base_PositionedAttr;
                       %statm_StatementAttr;
                       %statm_SelectionAttr;
                       %statm_SwitchAttr;>
<!ELEMENT statm:While (statm:hasCondition, statm:hasSubstatement*)>
<!ATTLIST statm:While %base_BaseAttr;
                      %base_PositionedAttr;
```

```
                    %statm_StatementAttr;
                    %statm_IterationAttr;
                    %statm_WhileAttr;>
<!ELEMENT statm:For (statm:hasForInit*, statm:hasCondition?,
                     statm:hasIncrement?, statm:hasSubstatement*)>
<!ATTLIST statm:For %base_BaseAttr;
                    %base_PositionedAttr;
                    %statm_StatementAttr;
                    %statm_IterationAttr;
                    %statm_ForAttr;>
<!ELEMENT statm:Do (statm:hasSubstatement*, statm:hasCondition)>
<!ATTLIST statm:Do %base_BaseAttr;
                   %base_PositionedAttr;
                   %statm_StatementAttr;
                   %statm_IterationAttr;
                   %statm_DoAttr;>
<!ELEMENT statm:Break EMPTY>
<!ATTLIST statm:Break %base_BaseAttr;
                      %base_PositionedAttr;
                      %statm_StatementAttr;
                      %statm_JumpAttr;
                      %statm_BreakAttr;>
<!ELEMENT statm:Continue EMPTY>
<!ATTLIST statm:Continue %base_BaseAttr;
                         %base_PositionedAttr;
                         %statm_StatementAttr;
                         %statm_JumpAttr;
                         %statm_ContinueAttr;>
<!ELEMENT statm:Return (statm:hasReturnValue?)>
<!ATTLIST statm:Return %base_BaseAttr;
                       %base_PositionedAttr;
                       %statm_StatementAttr;
                       %statm_JumpAttr;
                       %statm_ReturnAttr;>
<!ELEMENT statm:Goto (statm:jumpsToIdLabel)>
<!ATTLIST statm:Goto %base_BaseAttr;
                     %base_PositionedAttr;
                     %statm_StatementAttr;
                     %statm_JumpAttr;
                     %statm_GotoAttr;>
<!ELEMENT statm:Empty EMPTY>
<!ATTLIST statm:Empty %base_BaseAttr;
                      %base_PositionedAttr;
                      %statm_StatementAttr;
                      %statm_EmptyAttr;>
<!ELEMENT statm:CaseLabel (statm:hasCaseValue, statm:namesNextStatement)>
<!ATTLIST statm:CaseLabel %base_BaseAttr;
                          %base_PositionedAttr;
                          %statm_LabelAttr;
                          %statm_CaseLabelAttr;>
<!ELEMENT statm:DefaultLabel (statm:namesNextStatement)>
<!ATTLIST statm:DefaultLabel %base_BaseAttr;
                             %base_PositionedAttr;
                             %statm_LabelAttr;
                             %statm_DefaultLabelAttr;>
<!ELEMENT statm:IdLabel (statm:namesNextStatement)>
```

```
<!ATTLIST statm:IdLabel %base_BaseAttr;
                        %base_PositionedAttr;
                        %statm_LabelAttr;
                        %statm_IdLabelAttr;>
<!ELEMENT statm:hasBlock (statm:Block)>
<!ATTLIST statm:hasBlock>

<!ELEMENT statm:hasHandler (statm:Handler)>
<!ATTLIST statm:hasHandler>

<!ELEMENT statm:hasCatchParameter (statm:CatchParameter)>
<!ATTLIST statm:hasCatchParameter>

<!ELEMENT statm:hasCondition (%expressions;|struc:Object)>
<!ATTLIST statm:hasCondition>

<!ELEMENT statm:hasSubstatement ((%statements;|%expressions;|struc:NamespaceAlias|struc:Asm)
                                |(%classMembers;))>
<!ATTLIST statm:hasSubstatement>

<!ELEMENT statm:hasFalseSubstatement ((%statements;|%expressions;|
                                       struc:NamespaceAlias|struc:Asm)|
                                      (%classMembers;))>
<!ATTLIST statm:hasFalseSubstatement>

<!ELEMENT statm:hasCaseLabel (statm:CaseLabel)>
<!ATTLIST statm:hasCaseLabel>

<!ELEMENT statm:hasDefaultLabel (statm:DefaultLabel)>
<!ATTLIST statm:hasDefaultLabel>

<!ELEMENT statm:hasForInit ((%expressions;|statm:Empty)|(%classMembers;))>
<!ATTLIST statm:hasForInit>

<!ELEMENT statm:hasIncrement (%expressions;)>
<!ATTLIST statm:hasIncrement>

<!ELEMENT statm:hasReturnValue (%expressions;)>
<!ATTLIST statm:hasReturnValue>

<!ELEMENT statm:jumpsToIdLabel EMPTY>
<!ATTLIST statm:jumpsToIdLabel ref IDREF #REQUIRED>

<!ELEMENT statm:hasCaseValue (%expressions;)>
<!ATTLIST statm:hasCaseValue>

<!ELEMENT statm:namesNextStatement EMPTY>
<!ATTLIST statm:namesNextStatement ref IDREF #REQUIRED>

<!ELEMENT statm:hasTypeRep EMPTY>
<!ATTLIST statm:hasTypeRep ref IDREF #REQUIRED>

<!-- expr Elements  -->
<!ELEMENT expr:ExpressionList (%expressions;)*>
<!ATTLIST expr:ExpressionList %base_BaseAttr;
                              %base_PositionedAttr;
```

```
                            %expr_ExpressionListAttr;>
<!ELEMENT expr:New (expr:hasTypeRep, expr:hasNewPlacement?, expr:createsTypeRep,
                    expr:hasArguments?)>
<!ATTLIST expr:New %base_BaseAttr;
                   %base_PositionedAttr;
                   %expr_ExpressionAttr;
                   %expr_NewAttr;>
<!ELEMENT expr:Id (expr:hasTypeRep, expr:refersToName?)>
<!ATTLIST expr:Id %base_BaseAttr;
                  %base_PositionedAttr;
                  %expr_ExpressionAttr;
                  %expr_IdAttr;>
<!ELEMENT expr:TypeidType (expr:hasTypeRep, expr:takesTypeRep)>
<!ATTLIST expr:TypeidType %base_BaseAttr;
                          %base_PositionedAttr;
                          %expr_ExpressionAttr;
                          %expr_TypeidTypeAttr;>
<!ELEMENT expr:Conditional (expr:hasTypeRep, (%expressions;), (%expressions;), (%expressions;))>
<!ATTLIST expr:Conditional %base_BaseAttr;
                           %base_PositionedAttr;
                           %expr_ExpressionAttr;
                           %expr_ConditionalAttr;>
<!ELEMENT expr:SizeofType (expr:hasTypeRep, expr:takesTypeRep)>
<!ATTLIST expr:SizeofType %base_BaseAttr;
                          %base_PositionedAttr;
                          %expr_ExpressionAttr;
                          %expr_SizeofTypeAttr;>
<!ELEMENT expr:This (expr:hasTypeRep)>
<!ATTLIST expr:This %base_BaseAttr;
                    %base_PositionedAttr;
                    %expr_ExpressionAttr;
                    %expr_ThisAttr;>
<!ELEMENT expr:Throw (expr:hasTypeRep, (%expressions;)?)>
<!ATTLIST expr:Throw %base_BaseAttr;
                     %base_PositionedAttr;
                     %expr_ExpressionAttr;
                     %expr_ThrowAttr;>
<!ELEMENT expr:FunctionalConversion (expr:hasTypeRep, expr:convertsToTypeRep, expr:hasArguments)>
<!ATTLIST expr:FunctionalConversion %base_BaseAttr;
                                    %base_PositionedAttr;
                                    %expr_ExpressionAttr;
                                    %expr_FunctionalConversionAttr;>
<!ELEMENT expr:FunctionCall (expr:hasTypeRep, (%expressions;), expr:hasArguments)>
<!ATTLIST expr:FunctionCall %base_BaseAttr;
                            %base_PositionedAttr;
                            %expr_ExpressionAttr;
                            %expr_UnaryAttr;
                            %expr_FunctionCallAttr;>
<!ELEMENT expr:NamedConversion (expr:hasTypeRep, expr:convertsToTypeRep, (%expressions;))>
<!ATTLIST expr:NamedConversion %base_BaseAttr;
                               %base_PositionedAttr;
                               %expr_ExpressionAttr;
                               %expr_UnaryAttr;
                               %expr_NamedConversionAttr;>
<!ELEMENT expr:CastConversion (expr:hasTypeRep, expr:convertsToTypeRep, (%expressions;))>
<!ATTLIST expr:CastConversion %base_BaseAttr;
```

```
                              %base_PositionedAttr;
                              %expr_ExpressionAttr;
                              %expr_UnaryAttr;
                              %expr_CastConversionAttr;>
<!ELEMENT expr:PostIncDec (expr:hasTypeRep, (%expressions;))>
<!ATTLIST expr:PostIncDec %base_BaseAttr;
                          %base_PositionedAttr;
                          %expr_ExpressionAttr;
                          %expr_UnaryAttr;
                          %expr_PostIncDecAttr;>
<!ELEMENT expr:PreIncDec (expr:hasTypeRep, (%expressions;))>
<!ATTLIST expr:PreIncDec %base_BaseAttr;
                         %base_PositionedAttr;
                         %expr_ExpressionAttr;
                         %expr_UnaryAttr;
                         %expr_PreIncDecAttr;>
<!ELEMENT expr:TypeidExpr (expr:hasTypeRep, (%expressions;))>
<!ATTLIST expr:TypeidExpr %base_BaseAttr;
                          %base_PositionedAttr;
                          %expr_ExpressionAttr;
                          %expr_UnaryAttr;
                          %expr_TypeidExprAttr;>
<!ELEMENT expr:Indirection (expr:hasTypeRep, (%expressions;))>
<!ATTLIST expr:Indirection %base_BaseAttr;
                           %base_PositionedAttr;
                           %expr_ExpressionAttr;
                           %expr_UnaryAttr;
                           %expr_IndirectionAttr;>
<!ELEMENT expr:AddressOf (expr:hasTypeRep, (%expressions;))>
<!ATTLIST expr:AddressOf %base_BaseAttr;
                         %base_PositionedAttr;
                         %expr_ExpressionAttr;
                         %expr_UnaryAttr;
                         %expr_AddressOfAttr;>
<!ELEMENT expr:UnaryArithmetic (expr:hasTypeRep, (%expressions;))>
<!ATTLIST expr:UnaryArithmetic %base_BaseAttr;
                               %base_PositionedAttr;
                               %expr_ExpressionAttr;
                               %expr_UnaryAttr;
                               %expr_UnaryArithmeticAttr;>
<!ELEMENT expr:UnaryLogical (expr:hasTypeRep, (%expressions;))>
<!ATTLIST expr:UnaryLogical %base_BaseAttr;
                            %base_PositionedAttr;
                            %expr_ExpressionAttr;
                            %expr_UnaryAttr;
                            %expr_UnaryLogicalAttr;>
<!ELEMENT expr:SizeofExpr (expr:hasTypeRep, (%expressions;))>
<!ATTLIST expr:SizeofExpr %base_BaseAttr;
                          %base_PositionedAttr;
                          %expr_ExpressionAttr;
                          %expr_UnaryAttr;
                          %expr_SizeofExprAttr;>
<!ELEMENT expr:Delete (expr:hasTypeRep, (%expressions;))>
<!ATTLIST expr:Delete %base_BaseAttr;
                      %base_PositionedAttr;
                      %expr_ExpressionAttr;
```

```
                    %expr_UnaryAttr;
                    %expr_DeleteAttr;>
<!ELEMENT expr:Assignment (expr:hasTypeRep, (%expressions;), (%expressions;))>
<!ATTLIST expr:Assignment %base_BaseAttr;
                          %base_PositionedAttr;
                          %expr_ExpressionAttr;
                          %expr_BinaryAttr;
                          %expr_AssignmentAttr;>
<!ELEMENT expr:ArraySubscript (expr:hasTypeRep, (%expressions;), (%expressions;))>
<!ATTLIST expr:ArraySubscript %base_BaseAttr;
                              %base_PositionedAttr;
                              %expr_ExpressionAttr;
                              %expr_BinaryAttr;
                              %expr_ArraySubscriptAttr;>
<!ELEMENT expr:MemberSelection (expr:hasTypeRep, (%expressions;), (%expressions;))>
<!ATTLIST expr:MemberSelection %base_BaseAttr;
                               %base_PositionedAttr;
                               %expr_ExpressionAttr;
                               %expr_BinaryAttr;
                               %expr_MemberSelectionAttr;>
<!ELEMENT expr:Comma (expr:hasTypeRep, (%expressions;), (%expressions;))>
<!ATTLIST expr:Comma %base_BaseAttr;
                     %base_PositionedAttr;
                     %expr_ExpressionAttr;
                     %expr_BinaryAttr;
                     %expr_CommaAttr;>
<!ELEMENT expr:PointerToMember (expr:hasTypeRep, (%expressions;), (%expressions;))>
<!ATTLIST expr:PointerToMember %base_BaseAttr;
                               %base_PositionedAttr;
                               %expr_ExpressionAttr;
                               %expr_BinaryAttr;
                               %expr_PointerToMemberAttr;>
<!ELEMENT expr:BinaryArithmetic (expr:hasTypeRep, (%expressions;), (%expressions;))>
<!ATTLIST expr:BinaryArithmetic %base_BaseAttr;
                                %base_PositionedAttr;
                                %expr_ExpressionAttr;
                                %expr_BinaryAttr;
                                %expr_BinaryArithmeticAttr;>
<!ELEMENT expr:BinaryLogical (expr:hasTypeRep, (%expressions;), (%expressions;))>
<!ATTLIST expr:BinaryLogical %base_BaseAttr;
                             %base_PositionedAttr;
                             %expr_ExpressionAttr;
                             %expr_BinaryAttr;
                             %expr_BinaryLogicalAttr;>
<!ELEMENT expr:IntegerLiteral (expr:hasTypeRep)>
<!ATTLIST expr:IntegerLiteral %base_BaseAttr;
                              %base_PositionedAttr;
                              %expr_ExpressionAttr;
                              %expr_LiteralAttr;
                              %expr_IntegerLiteralAttr;>
<!ELEMENT expr:CharacterLiteral (expr:hasTypeRep)>
<!ATTLIST expr:CharacterLiteral %base_BaseAttr;
                                %base_PositionedAttr;
                                %expr_ExpressionAttr;
                                %expr_LiteralAttr;
                                %expr_CharacterLiteralAttr;>
```

```
<!ELEMENT expr:FloatingLiteral (expr:hasTypeRep)>
<!ATTLIST expr:FloatingLiteral %base_BaseAttr;
                               %base_PositionedAttr;
                               %expr_ExpressionAttr;
                               %expr_LiteralAttr;
                               %expr_FloatingLiteralAttr;>
<!ELEMENT expr:StringLiteral (expr:hasTypeRep)>
<!ATTLIST expr:StringLiteral %base_BaseAttr;
                             %base_PositionedAttr;
                             %expr_ExpressionAttr;
                             %expr_LiteralAttr;
                             %expr_StringLiteralAttr;>
<!ELEMENT expr:BooleanLiteral (expr:hasTypeRep)>
<!ATTLIST expr:BooleanLiteral %base_BaseAttr;
                              %base_PositionedAttr;
                              %expr_ExpressionAttr;
                              %expr_LiteralAttr;
                              %expr_BooleanLiteralAttr;>
<!ELEMENT expr:hasArguments (expr:ExpressionList)>
<!ATTLIST expr:hasArguments>

<!ELEMENT expr:hasNewPlacement (expr:ExpressionList)>
<!ATTLIST expr:hasNewPlacement>

<!ELEMENT expr:createsTypeRep EMPTY>
<!ATTLIST expr:createsTypeRep ref IDREF #REQUIRED>

<!ELEMENT expr:refersToName EMPTY>
<!ATTLIST expr:refersToName ref IDREF #REQUIRED>

<!ELEMENT expr:takesTypeRep EMPTY>
<!ATTLIST expr:takesTypeRep ref IDREF #REQUIRED>

<!ELEMENT expr:convertsToTypeRep EMPTY>
<!ATTLIST expr:convertsToTypeRep ref IDREF #REQUIRED>

<!ELEMENT expr:hasTypeRep EMPTY>
<!ATTLIST expr:hasTypeRep ref IDREF #REQUIRED>
```

Appendix B

DPML – Design Pattern Markup Language

```
<!-- Datatypes -->
<!ENTITY % Boolean "(true|false)">
<!ENTITY % AccessibilityKind "(private|protected|public|notPrivate|notProtected|notPublic)">
<!ENTITY % StorageClassKind "(static)">
<!ENTITY % FunctionKind "(normal|constructor|destructor)">

<!-- Elements -->
<!ELEMENT DesignPattern (Class*, TypeRep*)>
<!ATTLIST DesignPattern name CDATA #REQUIRED>

<!ELEMENT Class ((Base|Composition|Aggregation|Association)*, (Operation|Attribute)*)>
<!ATTLIST Class id           ID        #REQUIRED
                name         CDATA     #REQUIRED
                isAbstract   %Boolean; #IMPLIED
                isChangeable %Boolean; #IMPLIED>

<!ELEMENT Operation (defines?, (calls|creates)*, hasTypeRep, Parameter*)>
<!ATTLIST Operation id            ID                  #REQUIRED
                    name          CDATA               #REQUIRED
                    accessibility %AccessibilityKind; #IMPLIED
                    storageClass  %StorageClassKind;  #IMPLIED
                    kind          %FunctionKind;      #IMPLIED
                    isVirtual     %Boolean;           #IMPLIED
                    isPureVirtual %Boolean;           #IMPLIED>

<!ELEMENT Attribute (hasTypeRep)>
<!ATTLIST Attribute id            ID                  #REQUIRED
                    name          CDATA               #REQUIRED
                    accessibility %AccessibilityKind; #IMPLIED
                    storageClass  %StorageClassKind;  #IMPLIED>

<!ELEMENT Parameter (hasTypeRep)>
<!ATTLIST Parameter id   ID    #REQUIRED
                    name CDATA #REQUIRED>

<!ELEMENT Base EMPTY>
```

```
<!ATTLIST Base ref           IDREF               #REQUIRED
               accessibility %AccessibilityKind; #IMPLIED
               multiplicity  CDATA               #IMPLIED>

<!ELEMENT Composition EMPTY>
<!ATTLIST Composition ref           IDREF               #REQUIRED
                      accessibility %AccessibilityKind; #IMPLIED
                      multiplicity  CDATA               #IMPLIED>

<!ELEMENT Aggregation EMPTY>
<!ATTLIST Aggregation ref           IDREF               #REQUIRED
                      accessibility %AccessibilityKind; #IMPLIED
                      multiplicity  CDATA               #IMPLIED>

<!ELEMENT Association EMPTY>
<!ATTLIST Association ref           IDREF               #REQUIRED
                      accessibility %AccessibilityKind; #IMPLIED
                      multiplicity  CDATA               #IMPLIED>

<!ELEMENT defines EMPTY>
<!ATTLIST defines ref IDREF #REQUIRED>

<!ELEMENT calls EMPTY>
<!ATTLIST calls ref IDREF #REQUIRED>

<!ELEMENT creates EMPTY>
<!ATTLIST creates ref IDREF #REQUIRED>

<!ELEMENT TypeRep (TypeFormerType|TypeFormerArr|TypeFormerPtr|TypeFormerFunc)*>
<!ATTLIST TypeRep id ID #REQUIRED>

<!ELEMENT TypeFormerType EMPTY>
<!ATTLIST TypeFormerType ref IDREF #REQUIRED>

<!ELEMENT TypeFormerArr EMPTY>

<!ELEMENT TypeFormerPtr EMPTY>

<!ELEMENT TypeFormerFunc (hasReturnTypeRep, hasParameterTypeRep*)>

<!ELEMENT hasTypeRep EMPTY>
<!ATTLIST hasTypeRep ref IDREF #REQUIRED>

<!ELEMENT hasReturnTypeRep EMPTY>
<!ATTLIST hasReturnTypeRep ref IDREF #REQUIRED>

<!ELEMENT hasParameterTypeRep EMPTY>
<!ATTLIST hasParameterTypeRep ref IDREF #REQUIRED>
```

Appendix C

Summary

C.1 Summary in English

The main contributions of this work are summarized as follows. Most importantly, we define in detail a process for automatic tool-supported fact extraction and presentation and a schema for representing C++ source code entities and relations. We also present our reverse engineering framework that most ably supports our process. Furthermore, we show how tool interoperability can be achieved by using our schema instance conversions. To demonstrate the operability of our process and framework and the usability of our schema, we introduce two large applications. First, we present methods for recognizing design patterns in C++ source code; and second, we analyze the fault-proneness of open source software. Both applications are built on our general framework for reverse engineering.

Schema for the C++ programming language

This work was motivated by the observation that successful data exchange among reengineering tools is of crucial importance. This requires a common format so that the various tools (like front ends, metrics tools and clone detectors) can "talk" to each other. We designed a schema, called the *Columbus Schema for C++* [21; 26; 30], which captures information about source code written in the C++ programming language. The schema is modular, so it offers further flexibility for its extension or modification. The schema is fine-grained, representing practically every relevant fact about the source code so that logically equivalent source code can be generated from its instances.

The Columbus Schema for C++ satisfies some important requirements of an exchange format. It mirrors the low-level structure of the code as well as higher level semantic information (e.g. name resolution and semantics of types). Furthermore, the structure of the schema and the standard notation used (UML class diagrams) made its implementation and physical representation straightforward and, what is even more important, an API was very simple to prepare as well. Hence the Columbus Schema for C++ is a good candidate for exchanging information among tools of various types. This

is already supported by several studies, where the schema has been successfully used for data exchange. Perhaps the most important of these applications is GXL, a new standard for general information interchange in re- and reverse engineering.

C++ fact extraction process and framework

Extracting facts from small programs is simple enough and can be done even by hand. The real challenge is to analyze real-world software systems that consist of several million lines of code. The literature lacked methods and the community did not have tools with which such a complex task could be efficiently performed. We present a process [29] that lists five key steps which have to be done to successfully carry out a fact extraction task from C++ source code. The process deals with important points such as handling configurations, linking the schema instances, filtering the data obtained and converting it to other formats to facilitate data exchange.

The fact extraction process is supported by our framework, called the *Columbus Reverse Engineering Framework* [22–26; 28; 29] which was developed in an R&D project with the Nokia Research Center. The main motivation behind developing the framework was to create a toolset which supports fact extraction and provides a common interface for other reverse engineering tasks as well.

Design pattern recognition in C++ source code

A natural strategy of abstracting object-oriented programs is to represent them as a set of UML diagrams. While the automatic generation of UML diagrams from software code is supported by a number of reverse engineering tools, recognizing *design patterns* [34] is, currently, almost totally without advanced tool support. Design patterns are the most natural and useful assets when recovering the architectural design and the underlying design decisions from the software code. We present two methods for automatically recognizing design patterns from object-oriented (C++) code.

First, we present a method [27] and tool set for recognizing design patterns in C++ source code with the integration of Columbus and *Maisa* [52; 55]. The method combines the fact extraction capabilities of the Columbus reverse engineering framework with the clause-based pattern mining ability of Maisa. The C++ code was analyzed by Columbus, and we wrote a schema instance converter algorithm to export the collected facts to a form understandable to Maisa. This form is a clause-based design notation in PROLOG.

Second, we show a new approach to the problem of pattern detection which includes the detection of call delegations, object creations and operation redefinitions [3]. These are the elements that identify pattern occurrences more precisely. The pattern descriptions are stored in our new XML-based format, the *Design Pattern Markup Language* (*DPML*). This gives the user the freedom to modify the patterns, adapt them to his or her own needs, or create new pattern descriptions. The method was tested on four public-domain projects.

Analysis of the fault-proneness of open source software

Open source software systems are becoming evermore important these days. Many large companies are investing in open source projects and lots of them are also using this kind of software in their own work. As a consequence, many of these projects are being developed rapidly and quickly become very large. But because open source software is often developed by volunteers in their spare time, the quality and reliability of the code may be uncertain. Various kinds of code measurements can be quite helpful in obtaining information about the quality and fault-proneness of the code. We calculated the object-oriented metrics validated in [5; 7–9] for fault-proneness detection from the source code of the open source internet suite *Mozilla* [56] with the help of our framework. We then compared our results with those presented in [5]. One of our aims was to supplement their work with metrics obtained from a real-world software system. We also compared the metrics of seven versions of Mozilla to see how the predicted fault-proneness of the software changed during its development cycle [29].

Conclusion

With this work we achieved that it is now possible to reverse engineer the source code of large, real-world software systems written in the C++ programming language. What is more, we can do this without having to modify any source code, not even the makefiles or project files.

During the reverse engineering process our framework prepares a model of the analyzed C++ system according to a well-defined schema. Having such a schema enables the community and industry to improve existing reengineering tools and develop new ones that can seamlessly exchange information about the software system being studied. Our framework contains an application programming interface based on this schema with which the extracted information can easily be accessed and used. Output files in various formats can also be produced to promote tool interoperability.

C.2 Summary in Hungarian

Ebben a fejezetben a dolgozat fő eredményeit foglaljuk össze. A legfontosabb eredmény egy automatikus, eszközökkel támogatott tényfeltáró és bemutató eljárás részletes leírása, valamint egy séma definiálása a C++ programozási nyelvhez. Bemutatjuk még a visszatervező keretrendszerünket, amely messzemenően támogatja az eljárásunkat. Ezeken kívül bemutatjuk azt is, hogyan érhető el különböző eszközök között együttműködés a sémapéldányaink konverzióival. Hogy demonstráljuk az eljárásunk és keretrendszerünk működőképességét, valamint a sémánk használhatóságát, bemutatunk két nagy alkalmazást. Először a C++ forráskódban történő tervezési minta felismerésre mutatunk be módszereket, majd a nyílt forráskódú szoftverek hibára való hajlamosságát vizsgáljuk. Mindkét alkalmazás a keretrendszerünk segítségével valósult meg.

Séma a C++ programozási nyelvhez

Ezt a munkát az a felismerés motiválta, hogy rendkívüli a fontossága annak, hogy a különböző újratervező eszközök (mint például elemzők, metrikaszámítók és klón felismerők) adatokat tudjanak cserélni egymással. E cél eléréséhez szükség van egy közös formátumra, melynek segítségével kommunikálni tudnak egymással ezen eszközök. A szerző kidolgozott egy sémát, melynek a neve *Columbus Séma a C++*-hoz [21; 26; 30], amely C++ nyelven íródott forráskódról képes információkat ábrázolni. A séma moduláris, aminek következtében rugalmasan bővíthető és módosítható. Aprólékosan ábrázol minden fontos tényt a forráskódról úgy, hogy egy logikailag ekvivalens forráskód generálható a példányaiból.

A Columbus Séma a C++-hoz sok fontos, adatcserélő formátum igényt elégít ki. Képes finom részleteiben ábrázolni a kód struktúráját, de a magasabb szintű összefüggésekre is rámutat (pl. nevek feloldása és a típusok szemantikája). Ezen túl, a séma felépítése és a felhasznált standard jelölésmód (UML osztálydiagramok) lehetővé teszik a könnyű implementációját és fizikai ábrázolását, és ami még fontosabb, programozási interfészt is egyszerű volt hozzá készíteni. Ezekből kifolyólag a Columbus Séma a C++-hoz egy jó jelölt a különböző eszközök közötti adatcsere lebonyolítására. Ez már több tanulmánnyal is alá van támasztva, amelyek közül talán a legfontosabb a GXL alkalmazása.

C++ tényfeltáró eljárás és keretrendszer

Egy kis program tényfeltárása aránylag egyszerű és könnyedén elvégezhető kézzel is. Az igazi kihívást egy olyan valódi szoftver elemzése jelenti, amely több millió programsorból áll. Az irodalomból hiányoztak a megfelelő módszerek és a közösség nem rendelkezett megfelelő eszközökkel, hogy egy ilyen komplex feladatot el lehessen végezni. Bemutatunk egy eljárást [29], amely öt kulcsfontosságú lépésből áll, amelyek végrehajtásával sikeresen el lehet végezni egy C++ tényfeltáró eljárást. Az eljárás olyan fontos dolgokra tér ki,

mint például a konfigurációk kezelése, a sémapéldányok összefésülése, a feltárt tények szűrése, majd konvertálása, hogy elősegítsük az adatcserét a különböző eszközök között.

Egy a Nokia Kutatóközponttal együttműködésben lezajlott K+F projekt keretein belül kifejlesztettünk egy keretrendszert, a Columbus Visszatervező Keretrendszert [26], amely messzemenően támogatja a tényfeltáró eljárásunkat. A fő célkitűzésünk az volt, hogy létrehozzunk egy keretrendszert, amely támogatja a tényfeltárást és egy közös interfészt biztosít egyéb visszatervező feladatok elvégzésére is.

Tervezési minták felismerése C++ forráskódban

Az objektum-orientált programok absztrakt ábrázolásának egy természetes módja az UML diagramok [53] használata, de amíg a forráskódból több eszköz is képes UML diagramokat előállítani, addig a *tervezési minták* [34] felismerésére gyakorlatilag nincs szoftveres támogatás. Pedig ahhoz, hogy megbízhatóan rekonstruálni lehessen a forráskód architektúráját és a mögötte rejlő döntéseket, a tervezési minták felismerése elengedhetetlen. Két módszert ismertetünk a tervezési minták felismerésére C++ forráskódban.

Először bemutatunk egy módszert [27] és eszközkészletet tervezési minták felismerésére a Columbus és a Maisa [52; 55] szoftverek integrációjával. Ez a módszer kibővíti a Columbus keretrendszer tényfeltáró képességeit a Maisa mintafelismerő képességével. A C++ kódot először analizáltuk a Columbus segítségével, és készítettünk egy sémapéldány konvertáló algoritmust, amely adatokat gyárt a Maisa bemeneti formátumában, amely egy PROLOG-jellegű nyelv.

A másik módszer egy új megoldást ad a tervezési minta keresés problémájára [3], amely magában foglalja a függvényhívások, objektumlétrehozások és operáció felüldefiniálások felismerését is. Ezek azok az elemek, amelyekkel képesek vagyunk pontosabban meghatározni a mintapéldányokat. A keresett minták az általunk definiált új, XML-alapú, *Design Pattern Markup Language* (*DPML*) nevű nyelven vannak leírva. Ezáltal a mintaleírásokat szabadon lehet módosítani, hozzáilleszteni bizonyos helyzetekhez illetve akár új leírásokat is létre lehet hozni. A módszert négy szabadon elérhető szoftveren teszteltük.

Nyílt forráskódú szoftverek hibára való hajlamosságának vizsgálata

Napjainkban a nyílt forráskódú szoftverek egyre fontosabbakká válnak. Sok nagy cég fektet be nyílt forráskódú projektekbe, és sok közülük használja is ezeket a szoftvereket a mindennapi munka során. Következésképpen, sok ilyen projekt rohamosan fejlődik és gyorsan nő a mérete. De mivel a nyílt forráskódú szoftvereket általában önkéntesek fejlesztik a szabad idejükben, a forráskód minősége és megbízhatósága kétséges lehet. Különböző kódmérések igazán hasznosak lehetnek, hogy többet lehessen tudni a kód minőségéről és hibára való hajlamosságáról. A keretrendszerünk segítségével kiszámítottuk

az irodalomban [5; 7–9] ellenőrzött, a forráskód hibára való hajlamát előjelző metrikákat a nyílt forráskódú, *Mozilla* [50; 56] nevű internetes szoftver forráskódjából. Ezután összehasonlítottuk a kapott eredményeket a [5]-ben publikáltakkal. Az egyik célkitűzésünk az volt, hogy kiegészítsük az eredményeiket egy valós világból vett szoftver mérési eredményeivel. Ezen kívül összehasonlítottuk a Mozilla hét különböző verziójának (1.0–1.6) mért értékeit hogy megvizsgálhassuk hogyan változott a hibára való hajlamossága a fejlesztése során.

Konklúzió

Ezzel a munkával elértük, hogy ma már lehetséges a nagy, valós világból vett, C++ programozási nyelven írt szoftverrendszerek forráskódjának a visszatervezése. Mitöbb, ehhez nem kell módosítani sem a forráskódot, sem a makefile- vagy projekt fájlokat.

A tényfeltáró eljárás során a keretrendszerünk elkészíti az analizált C++ rendszer egy modelljét egy jól meghatározott sémának megfelelően. Egy ilyen séma birtokában a közösség és az ipar számára lehetőség nyílik az eszközeik továbbfejlesztésére, valamint újak létrehozására, amelyek már zökkenőmentesen képesek lesznek a tanulmányozott rendszerről információt cserélni. A keretrendszerünk tartalmaz egy a sémára alapozott programozói interfészt is, amellyel a feltárt tények könnyedén elérhetők és felhasználhatók. Különböző formátumú kimeneti fájlok is készíthetők, hogy még inkább elősegítsük az eszközök közötti együttműködést.

Bibliography

[1] AllFusion Component Modeler (formerly Paradigm Plus) Homepage. `http://www3.ca.com/Solutions/ Product.asp?ID=1003`.

[2] M.N. Armstrong and C. Trudeau. Evaluating Architectural Extractors. In *Proceedings of WCRE'98*, pages 30–39, October 1998.

[3] Zsolt Balanyi and Rudolf Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, September 2003.

[4] Jagdish Bansiya. DP++ is a tool for C++ programs. In *Dr. Dobb's Journal*, June 1998.

[5] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. In *IEEE Transactions on Software Engineering*, volume 22, pages 751–761, October 1996.

[6] Bell Canada Inc., Montréal, Canada. *DATRIX – Abstract semantic graph reference manual*, version 1.4 edition, May 2000.

[7] Lionel C. Briand, Walcelio L. Melo, and Jürgen Wüst. Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects. In *IEEE Transactions on Software Engineering*, volume 28, pages 706–720, July 2002.

[8] Lionel C. Briand and Jürgen Wüst. Empirical Studies of Quality Models in Object-Oriented Systems. In *Advances in Computers*, volume 56, September 2002.

[9] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. In *The Journal of Systems and Software*, volume 51, pages 245–273, 2000.

[10] Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. In *Master's thesis*. Department of Computer Engineering, North Carolina State University, 1996.

[11] Bugzilla for Mozilla. `http://bugzilla.mozilla.org`.

[12] Eduardo Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, 10(8):45–52, 1998.

[13] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *IEEE Transactions on Software Engineering*, volume 24, pages 682–693, September 1998.

[14] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. A c++ data model supporting reachability analysis and dead code detection. *IEEE Transanctions on Software Engineering*, 24(9):682–694, 1998.

[15] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object-Oriented Design. In *IEEE Transactions on Software Engineering 20,6(1994)*, pages 476–493, 1994.

[16] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. In *IEEE Software 7*, pages 13–17, January 1990.

[17] Jörg Czeranski, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödereder, Daniel Simon, Yan Zhang V, Jean-Francois Girard, and Martin Würthner. Data exchange in Bauhaus. In *WCRE*, pages 293–295, 2000.

[18] Thomas R. Dean, Andrew J. Malton, and Ric Holt. Union Schemas as a Basis for a C++ Extractor. In *Proceedings of WCRE'01*, pages 59–67, October 2001.

[19] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization. In *Proceedings of WCRE'99*, 1999.

[20] J Ebert, R Gimnich, H H Stasch, and A Winter. GUPRO – Generische Umgebung zum Programmverstehen, 1998.

[21] Rudolf Ferenc and Árpád Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, March 2002.

[22] Rudolf Ferenc and Árpád Beszédes. Az Objektumvezérelt Szoftverek Elemzése. In *VIII. Országos (Centenáriumi) Neumann Kongresszus Elõadások és Összefoglalók*, pages 463–474. Neumann János Számítógép-tudományi Társaság, October 2003.

[23] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Extracting Facts with Columbus from C++ Code. In *Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 4–8, March 2004.

[24] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Fact Extraction and Code Auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, page 513. IEEE Computer Society, September 2004.

[25] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. *Tools for Software Maintenance and Reengineering*, chapter Extracting Facts with Columbus from C++ Code, pages 16–31. Franco Angeli Milano, 2004.

[26] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.

[27] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybernetica*, 15:669–682, 2002.

[28] Rudolf Ferenc, Ferenc Magyar, Árpád Beszédes, Ákos Kiss, and Mikko Tarkiainen. Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 16–27. University of Szeged, June 2001.

[29] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, September 2004.

[30] Rudolf Ferenc, Susan Elliott Sim, Richard C Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a Standard Schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 49–58. IEEE Computer Society, October 2001.

[31] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. In *IBM Systems Journal*, volume 36, pages 564–593, November 1997.

[32] F. Fioravanti and P. Nesi. A Study on Fault-Proneness Detection of Object-Oriented Systems. In *Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 121–130, March 2001.

[33] R. Fiutem G. Antoniol and L. Cristoforetti. Using Metrics to Identify Design Patterns in Object-Oriented Software. In *Proceedings of the Fifth International Symposium on Software Metrics (METRICS98)*, pages 23–34, November 1998.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[35] Michael W. Godfrey and Eric H. S. Lee. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, pages 15–23, June 2000.

[36] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design patterns identification. In *Proceedings of IJCAI Workshop on Modelling and Solving Problems with Constraints*, pages 57–64, August 2001.

[37] The GXL Homepage. `http://www.gupro.de/GXL`.

[38] Ric Holt, Ahmed E. Hassan, Bruno Laguë, Sébastien Lapierre, and Charles Leduc. E/R Schema for the Datrix C/C++/Java Exchange Format. In *Proceedings of WCRE'00*, November 2000.

[39] Ric Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, November 2000.

[40] Holt, R. C. An Introduction to TA: The Tuple-Attribute Language. `http://plg.uwaterloo.ca/~holt/papers/ta-intro.doc`.

[41] International Standards Organization. *Programming languages – C++*, ISO/IEC 14882:1998(E) edition, 1998.

[42] IBM Jikes Project. `http://oss.software.ibm.com/developerworks/opensource/jikes`.

[43] Christian Kraemer and Lutz Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*, pages 208–215, November 1996.

[44] The LEDA Homepage. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[45] Timothy C. Lethbridge. Requirements and Proposal for a Software Information Exchange Format (SIEF) Standard. `http://www.site.uottawa.ca/~tcl/papers/sief/standardProposal.html`.

[46] Timothy C. Lethbridge. The dagstuhl middle model: An overview. volume 94, pages 7–18. ELSEVIER, Electronic Notes in Theoritical Computer Science, 2003.

[47] E Mamas and K Kontogiannis. Towards Portable Source Code Representations Using XML. In *Proceedings of WCRE'00*, pages 172–182, November 2000.

[48] J. Mayrand and F. Coallier. System Acquisition Based on Product Assessment. In *Proceedings of ICSE'96*, 1996.

[49] K. Mehlhorn and S. Naeher. LEDA: A Platform for Combinatorial and Geometric Computing. In *Cambridge University Press*, 1997.

[50] The Mozilla Homepage. `http://www.mozilla.org`.

[51] Hausi A Müller, Kenny Wong, and Scott R Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Proceedings of ACFAS*, 1994.

[52] L. Nenonen, J. Gustafsson, J. Paakki, and A.I. Verkamo. Measuring Object-Oriented Software Architectures from UML Diagrams. In *Proceedings of the 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 87–100, 2000.

[53] Object Management Group Inc. *OMG Unified Modeling Language Specification*, 1.3 edition, 1999.

[54] Object Management Group Inc. *OMG XML Metadata Interchange (XMI) Specification*, 1.1 edition, 2000.

[55] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A.I. Verkamo. Software Metrics by Architectural Pattern Mining. In *Proceedings of he International Conference on Software: Theory and Practice (16th IFIP World Computer Congress).*, pages 325–332, 2000.

[56] Christian Robottom Reis and Renata Pontin de Mattos Fortes. An Overview of the Software Engineering Process and Tools in the Mozilla Project. In *Proceedings of the Workshop on Open Source Software Development*, pages 155–175, February 2002.

[57] The Rigi Homepage. `http://www.rigi.csc.uvic.ca`.

[58] Claudio Riva, Michael Przybilski, and Kai Koskimies. Environment for Software Assessment. In *Proceedings of ECOOP'99*, 1999.

[59] Sander, G. VCG Overview. `http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html`.

[60] Susan Elliott Sim and Margaret-Anne D. Storey. A structured demonstration of program comprehension tools. In *WCRE*, pages 184–193, 2000.

[61] The Source-Navigator IDE Homepage. `http://sourcenav.sourceforge.net`.

[62] The StarOffice Homepage. `http://www.sun.com/software/star/staroffice`.

[63] Margaret-Anne Storey, Casey Best, and Jeff Michaud. Shrimp views: An interactive environment for exploring java programs. In *9th International Workshop on Program Comprehension (IWPC 2001)*, pages 111–112. IEEE Computer Society, 2001.

[64] Bjarne Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley Professional, 2000.

[65] A Taivalsaari and S Vaaraniemi. TDE: Supporting Geographically Distributed Software Design with Shared, Collaborative Workspaces. In *Proceedings of CAiSE'97*, LNCS 1250, pages 389–408. Springer Verlag, 1997.

[66] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Columbus Schema for C/C++ Preprocessing. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 75–84. IEEE Computer Society, March 2004.

[67] Ping Yu, Tarja Systä, and Hausi Müller. Predicting Fault-Proneness using OO Metrics: An Industrial Case Study. In *Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 99–107, March 2002.