A C++ Refactoring Browser and Method Extraction

Marian Vittek[1], Peter Borovansky[1], and Pierre-Etienne Moreau[2]

[1] FMFI, Comenius University, Mlynska dolina, 842 15 Bratislava
Slovakia
{vittek,borovan}@fmph.uniba.sk
[2] LORIA-INRIA, BP 239, 54506 Vandœuvre-lès-Nancy
France
moreau@loria.fr

Abstract. This paper presents a refactoring tool for C++. Its implementation illustrates the main difficulties of automated refactoring raising in this case from the preprocessor and from the complexity of the language. Our solution, using a back-mapping preprocessor, works in the presence of complex preprocessor constructions built upon file inclusions, macro expansions and conditional compilations. Refactorings are computed after full preprocessing and parsing of target programs, hence, they are based on the same level of program understanding as performed by compilers. The paper illustrates the main ideas of our approach on the example of *Extract Method* refactoring. [3].

1 Introduction

Maintenance of large legacy software systems is a hard task. *Refactoring* [9, 10, 15] is a promising methodology helping developers in this work. Refactoring is a software development and maintenance process where the source code is changed "in such a way that it does not alter the external behavior of the code yet improve its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs" [9].

For example, renaming of a global variable iii to $fileIndex$ on all its occurrences is a refactoring. Replacing on all its occurrences means that only occurrences of this global variable will be renamed. There may be several local variables iii (or many class members named iii) which will not be renamed because they are not *linked* to the global iii. In opposition to the full text replacement this is a kind of *minimal* or *required* renaming, making only necessary modifications. Such refactoring improves the quality of the code because it makes it more readable.

Refactoring browsers are software tools helping maintainers in performing refactorings. In the context of refactoring browsers, the word refactoring is used as a noun to describe a simple elementary behavior preserving source transformation. When using a refactoring tool, a human maintainer only selects the required transformation (such as *rename variable*) and its input parameters (new name for the variable) and the tool performs all necessary source modifications. The tool shall guarantee that those modifications do not change the behavior of the program and, hence, does not introduce any new bug. Use of an automatic tool allows to perform massive changes in source code

[3] This work was supported by Agency for Promotion Research and Development under the contract No. APVV-20-P04805.

quickly and safely. Refactoring tools usually perform a fixed set of refactorings identified by short schematic names, such as *Rename Variable, Move Field, Move Method, Encapsulate Field, Pull Up Field/Method, Push Down Field/Method, Extract Method*, etc. Among them, the *Extract Method* has a particular position from the point of view of evaluation of refactoring tools. This refactoring consists of extraction of a piece of code into a newly created method with automatically generated parameters and return value. Parameters and the return value are computed after a static analysis of the program and the quality of this analysis often indicates the quality of the whole refactoring tool. The method extraction also belongs to the most used refactorings.

In this paper, we present *Xrefactory C++*, a tool performing refactorings on C++ programs. The implementation evolved from our previous works on C language [25]. Our approach is using a back-mapping preprocessor and, while computing the refactorings, it performs full preprocessing and parsing of C++ programs. At this moment, our tool performs only a limited set of refactorings, namely all kinds of renaming, refactorings for adding, removing and moving method parameters and the method extraction. Others are in progress.

2 Related Works

The concept of refactoring has been discussed during more than a decade [9, 15, 19, 21]. In chronological order, refactoring has been investigated mainly in the context of Smalltalk and C++ programming languages without experimental implementations. Later, the Smalltalk refactoring browser [20] has been developed and was probably the world's first automated refactoring tool. The importance of refactoring was strengthen by *extreme programming* methodology [5, 14] as one of its basic rules. Independently, similar works on program transformations were held in the context of term rewriting [18, 23]. After apparition of Martin Fowler's book [9] numerous implementations of refactoring browsers for Java emerged, among them Intellij IDEA and Eclipse. Refactoring techniques heavily depend on the underlying programming language and are usually studied and implemented separately for each language. The refactoring of C/C++ has been discussed during several years [8, 19, 22]. However, problems with the language complexity and the preprocessor caused that C/C++ refactoring tools are still rare, and not widely accepted. The preprocessor is not part of the language in traditional sense. It does not enter into its grammar and it makes standard techniques very difficult to apply. Despite the practical motivations, there is only a small research activity in the area of C++ refactoring. Eclipse [1] team is working on refactoring support for the CDT module which is currently limited to a restrictive form of renaming. At the author best knowledges, the real state of the art is represented by two tools: *Visual SlickEdit* [3] and *Ref++* [2]. The first one is a self standing Integrated Development Environment (IDE) which incorporates C++ refactoring features. The second one is a plugin to MSVC++ environment. They are both approaching C++ refactoring in a practical manner implementing a large number of refactorings (between 10 and 20), however, they are both working correctly only in rather simple common circumstances. Those tools offer a usable solution while not going too deeply into the program structure and, in particular, they do not really analyze problems introduced by the combination of the language and

the preprocessor. For example, at this time, none of those tools is able to perform correct refactorings at places where an #if preprocessor directive with an #else branch is present.

A deeper comparison of our approach and the mentioned two tools is difficult because both tools are commercial and there are only few informations available about their internal structure and implementation. Anyway, from the external behavior, we can see that our approach is based on deeper understanding of the source code than provided by those tools. As we will present in the paper, our tool proceeds correctly all features of the C++ language as well as all preprocessor constructions including complex combinations of #if-#else directives. On the other side, due to the complexity of the implementation, at the moment, our tool is implementing much smaller set of refactorings.

There is also a number of other related works needed to be mentioned at this place, even if they are not directly concerned by C++ refactoring. Alejandra Garrido and Ralph Johnson work on a C refactoring browser at University of Illinois at Urbana-Champaign [11–13]. Their approach is focused on correct handling of all preprocessor constructions, including very complex conditional directives, however, few syntactical restrictions on the usage of those directives are applied. Their tool is relied to the C language and the solution is not directly extensible to C++. In Berkeley, Bill McCloskey and Eric Brewer [17] work on a C-like language, where preprocessor directives will be defined at the language level instead of being purely textual. They suppose that each C program can be translated into this language and then easily refactored. Several other projects are dealing with preprocessor while not focused on refactoring. Semantic Designs is working on a set of source understanding and transformation tools for variety of languages including C and C++ [4]. An interesting tool focused on porting C++ programs from one platform to another is developed by D. Waddington and B. Yao from Lucent Technologies [7]. Those projects are, in general, incorporating preprocessor constructions in the Abstract Syntax Tree (AST), or they are incorporating them directly into the grammar of the language. Last but not least, a number of practical problems connected to the preprocessor has been seriously examined in independent works focused on source understanding tools [6, 24].

3 C++ Refactoring and the Preprocessor

The C++ language evolved from C by accepting wide range of extensions, including object oriented classes, (multiple) inheritance, overloaded methods and operators, virtual methods, namespaces, exceptions, templates, etc. An unwanted side effect of those generous extensions is the difficulty of parsing, understanding and refactoring C++ programs. Moreover, a usual C++ program is not written in C++. It contains *preprocessor directives* which are not part of C++ grammar.

Preprocessor is a serious obstacle in development of a refactoring tool. One possible approach how to deal with preprocessor directives is their incorporation into the grammar and the AST and development of a parser working with those extensions. Unfortunately, the C++ parser itself is very complex and its development is on the limits of many companies, a direct combination with the preprocessor and development of a parser for such mixed language seems unrealistic for us. For this reason, we are using a different approach in our implementation. We are using a standard preprocessor,

parser and AST. Similarly to [6], our preprocessor is extended to generate an additional back-mapping information allowing to trace each character of the code. Beyond the usual preprocessed code, the preprocessor generates a table determining for each character of preprocessed code, from which place of the original source code it comes. Refactorings are then computed on the preprocessed AST and the necessary source transformations are backmapped to the original code. This approach is solving majority of problems related to the parsing of C++, because we can use a standard parser developed for compilers. With a small effort, it also solves problems related to macro expansions and file inclusions. The real problem is the presence of #if-#else directives. This directive is basically used to trigger different fragments of code in or out of the compilation depending on an external configuration. Different configurations are represented by different initial setting of predefined macros. In order to be able to correctly understand the whole program (i.e. both positive and negative branches of #if directives), we have decided to parse the source code several times during a single refactoring. Each parsing is performed with different initial macro settings. The considered initial macro settings are entered by the user and are supposed to cover all possible compilations of the project. The refactoring is computed after having performed all parsings. The way, how the resulting refactoring is combined from the parsings is specific for each particular refactoring. For example, in the case of symbol renaming and parameter manipulations, the resulting refactoring is basically a union of all required modifications (renaming) from all passes. The situation is more difficult in the case of method extraction.

In the rest of the paper, we will illustrate our implementation by describing in details the implementation of the method extraction. We prefer to explain our approach by introducing this single refactoring instead of describing the whole tool. We feel that in this way, we will illustrate better the main ideas on which the tool is built as well as the overall complexity of the implementation.

4 Simple Code Extraction

Extraction of a method is a simple and intuitive program transformation, a kind of intelligent 'cut and paste'. For example, lets take the following program computing the n-th Fibonacci number for a given parameter:

```
ln  1: int main(int argc, char **argv) –
ln  2:     int i,n,x,y,t;
ln  3:     sscanf(argv[1], "%d", &n);
ln  4:     x=0; y=1;
ln  5:     for(i=0; i¡n; i++) –
ln  6:         t=x+y; x=y; y=t;
ln  7:     ˝
ln  8:     printf("%d-th fib == %d“n", n, x);
ln  9:     return(0);
ln 10: ˝
```

When a refactoring tool is asked to extract the code between lines 4 and 7 into a method (say fib), it replaces the original code by:

```
ln  1: static int fib(int n) –
```

```
ln  2:     int t, y, x, i;
ln  3:     x=0; y=1;
ln  4:     for(i=0; i¡n; i++) –
ln  5:         t=x+y; x=y; y=t;
ln  6:     ″
ln  7:     return(x);
ln  8: ″
ln  9:
ln 10: int main(int argc, char **argv) –
ln 11:     int i,n,x,y,t;
ln 12:     sscanf(argv[1], ”%d”, &n);
ln 13:     x = fib(n);
ln 14:     printf(”%d-th fib == %d“n”, n, x);
ln 15:     return(0);
ln 16: ″
```

Determining which variable should be a parameter of the new method is made automatically. From the implementation point of view, it requires static analysis of the method, in particular of its local variables. The analysis classifies each local variable into one of five categories: *none, local, in, out, in-out* saying that it is respectively not concerned by the extraction, will become a local variable of the new method, will become an input parameter (passed by value), output parameter, or input/output parameter (passed by address). Later, if there is only one output variable, (and it has a base type), it may be reclassified to *return value*.

The analysis of local variables is similar to variable lifetime analysis performed by compilers. For each variable, the method is transformed into a control flow diagram. The diagram is examined and all usages of each variable are watched. The tool is especially examining whether a value assigned outside the extracted block is used inside the block, and vice-versa. If such a control flow is discovered, the corresponding flag is set. After examination of all possible control flows, resulting flags are evaluated and the variable is classified to one of the above five categories. There is one more problem needed to be considered. Let's take the following simple function writing numbers from 0 to 9 and let us suppose that we are going to extract the single line 5 into a new method.

```
ln  1: void fun() –
ln  2:     int i=0;
ln  3:     for(int j=0; j¡10; j++) –
ln  4:         /* block begin */
ln  5:         cout ¡¡ i++;
ln  6:         /* block end */
ln  7:     ″
ln  8: ″
```

In this case, the above analysis indicates that there is a value of the variable i assigned outside the block which is used inside and there is no value of the variable i assigned inside the block which is used outside. Logically, this would give a classification for i as being an *input* variable. However, due to the loop re-entrance, this variable has to be classified as *input/output*. At the time, this benchmark has caused problems to many

professional refactoring browsers. In our implementation, we have solved this problem by introducing a new flag, indicating whether, in the given control flow, a value has been passed outside the block and reentered into the block and then reused inside. The presence of address parameters is a particularity of C++ compared to Java and C. Their presence in a method requires another small refinement. When watching an address parameter, all leaving points of the method has to be considered as places where it is (potentially) used. This is because it may carry out resulting values.

After having classified all variables, the actual extraction of the code is just a question of text editing. The tool performs moving of the extracted text, generates the header and the footer of the method and its invocation at the place from where it was extracted. When a method is a class member and its body is not inside the class, a declaration of the method into the corresponding class definition has to be generated too.

So far, we did not consider the preprocessor. Possible presence of preprocessor directives complicates nearly all parts of the implementation. In the following text we will discuss the complications and the solutions, we have adopted, for each of preprocessor directives separately.

5 Macros

Macros are defined with the #define preprocessor directive. They allow to textually replace *macro usages* into *macro bodies*, which are usually pieces of code. During the method extraction, a question arises whether to perform extraction after the macro expansion or before. In Xrefactory, we are using in fact a combination of both. It is obvious that the analysis and classification of local variables has to be done after macro expansion. Otherwise the analysis may be wrong. For example, if we take a fragment:

```
#define ASSIGN(x,v) –x=v;"
...
   int i;
   ASSIGN(i,5);
```

without the macro expansion, we do not know that the variable i is used as an l-value in the fragment.

On the other side, the new extracted method must be unpreprocessed. It corresponds to the intuition that the code written in a file is not preprocessed and the new method should be written in the same manner as it was written by the original developer. We have solved those two points in a direct ad-hoc way, the variable analysis is performed after the full preprocessing of the source code, while the actual text extraction is performed on the original unpreprocessed code. Xrefactory actually implements a real textual cut and paste of the original code moving the method body as text. This solution protects original formatting and unpreprocessed structure. It means also that macro usages are not expanded in the extracted method (even if they were during the variable analysis).

A similar situation occurs during the generation of the *header* and the *footer* of the new method. In our terminology, the header is the beginning of the new method defining parameters and those local variables carried from outer environment. In the introductory example, that were the lines 1 and 2 of the method fib. The footer is the piece of code

assigning the return value (if any). In our introductory example, that were the lines 7 and 8. The header and the footer are pieces of code entirely generated by Xrefactory, it may seem, that they will be generated from an internal program representation. However, it may happen that, for example, definitions of variables present in the header used macros. The typical situation is definition of arrays using macros in their dimensions. Let's take a code containing an array a and let us suppose that the array is classified as a *local* variable for the extracted method. So, the situation in the original code is:

```
#define MAX˙VALUES 1000
...
    int i, j, a[MAX˙VALUES];
...
```

It is logical that the generated header should be:

```
void extracted() –
  int a[MAX˙VALUES];
```

instead of:

```
void extracted() –
  int a[1000];
```

In other words, when the original definition of the variable contains macros, it is expected that the definition in the generated header will contain macros too. For this reason, Xrefactory composes also definitions of all parameters by a copy-pasting of corresponding pieces of the original text. From the implementation point of view, it requires that the parser remembers source positions of corresponding syntactic categories, such as *declaration specifiers* and *declarator* of the parsed text. Moreover, because the parser acts on the preprocessed code, those positions has to be back-mapped to positions in the unpreprocessed code and corresponding pieces of code are taken from there.

6 File Inclusion

The include directive allows to insert an entire file into a particular place of source code. In a usual case, this directive does not represent a particular complication for method extraction. Inclusion is, in its nature, a very similar operation to macro expansion. One can imagine that the whole included file is just a body of a long macro which is expanded at the given place. Effects of the file inclusion are then handled by the same techniques as macro expansions, i.e. by the back-mapping preprocessor. Note also, that the include directive rarely occurs in the body of a method, hence, it only rarely interfere with extraction.

7 Conditional Compilation

The C/C++ preprocessor allows to include or exclude some part of source code depending on a condition evaluated in compile time. The mechanism is implemented

via #if directive and looks like a usual if statement. Conditional compilation is the main difficulty introduced by the preprocessor to refactoring tools. The difficulty comes from usages where the #if directive is used in combination with platform or compiler specific features. Let's take, for example, the code:

```
ln  1: #if defined(¨WIN32)
ln  2: #include ¡windows.h¿
ln  3: #define PMACRO(x) Jbox(x++)
ln  4: #else
ln  5: #define PMACRO(f,x) printf(f,x)
ln  6: #endif

ln  8: ...
ln  9: #if defined(¨WIN32)
ln 10:     PMACRO(i);
ln 11: #else
ln 12:     PMACRO("%d", i);
ln 13: #endif
ln 14: ...
```

This example is a bit artificial, however it perfectly illustrates two problems related to conditionals. The first problem is that there may be several dependent conditionals and the program may be unparsable if they are not processed in the same way. The second problem is that when extracting lines $9 - 13$, then two analysis on two different platforms lead to different classifications of the parameter i. On Windows platform, it will be classified as *input/output* parameter and on other platforms as *input* parameter. The problem is: in which way to pass this variable to the extracted method so that it works under both systems?

In our implementation, we have solved both problems by performing multiple passes over the source code. The number of passes as well as the initial configurations are fully specified by the user. The user has to specify all combinations of initial macro definitions which occur in various compilations of the project. For each such combination (for each preprocessor pass), the source is preprocessed, parsed and the static analysis of variables is performed. Variables are classified (to be *input, output, etc.* parameters) as described in the previous section. After all passes, those informations are combined into a final resulting classification. The computation of the final classification is quite intuitive, it is implemented by a binary operation $reclassify$ starting from the initial classification and merging it with all following passes. The table 1 shows all possibilities of variable reclassification. After all passes a variable obtains its final classification (which is the most general of all passes) and the corresponding header of the new method is generated upon this resulting classification. So, finishing our example, the variable i would be classified as being an *input/output* parameter after both passes.

This solution works well when the variable occurs in all passes. However, a variable can be missing in some passes. For example, let us take the situation:

```
ln  1: void drawCircle(Point c, int d
ln  2: #if defined(USE˙COLORS)
ln  3:                 , Color color
```

	n	l	i	o	io
n	n	l	i	o	io
l	l	l	i	o	io
i	i	i	i	io	io
o	o	o	io	o	io
io	io	io	io	io	io

Tab. 1. Reclassification of variables after two preprocessor passes. The figure shows the resulting classification of a variable depending on its classifications in the first and the second pass. Abbreviations for classifications are: n=$none$, l=$local$, i=$input$, o=$output$, io=$input/output$.

```
ln  4: #endif
ln  5:                ) –
ln  6: ...
ln  7: #if defined(USE˙COLORS)
ln  8:     setColor(color);
ln  9: #endif
ln 10: ...
ln 11: ˝
```

In this example if we are extracting a block containing only one line, namely the line 8 with setColor invocation, there will be (at least) two preprocessor passes. A pass where USE˙COLORS is defined and another where it is not. In the first pass, (when USE˙COLORS is defined) there is a variable color, which is classified as an *input* parameter. When USE˙COLORS is not defined, the variable does not exist at all, so the above reclassification is not possible. In our implementation, we have considered two solutions for merging both passes in such situation. The first possibility is to classify the variable according to the existing pass and to ignore another pass. The second solution is to note that the variable exists only in some passes and to generate additional #if directive around its definition. Those approaches would lead respectively to following extractions, the first approach would generate:

```
void extracted(Color color) –
    setColor(color);
˝
...
#if defined(USE˙COLORS)
  extracted(color);
#endif
```

and the second approach:

```
void extracted(
#if defined(USE˙COLORS)
              Color color
#endif
              ) –
    setColor(color);
```

```
~
...
#if defined(USE˙COLORS)
  extracted(
#if defined(USE˙COLORS)
            color
#endif
           );
#endif
```

In the last example, a more careful evaluation of guarding #if directives may remove the nested conditional. It seems to us that the first approach generates a more appropriate extraction for human maintainers. Its risky point is that the definition of the variable may be unparsable in some circumstances. For example, if the type Color is defined only in passes when USE˙COLORS was predefined. The situation may be even worse, in the following example. Let's consider that we extract also the guarding #if directives, i.e. we extract a block of lines $7-9$ instead of the single line 9. To compare both approaches in this case, we will get the following extraction for the first approach:

```
void extracted(Color color) –
#if defined(USE˙COLORS)
    setColor(color);
#endif
~
...
  extracted(color);
```

and for the second:

```
void extracted(
#if defined(USE˙COLORS)
               Color color
#endif
               ) –
#if defined(USE˙COLORS)
    setColor(color);
#endif
~
...
  extracted(
#if defined(USE˙COLORS)
            color
#endif
           );
```

In the first approach, the invocation of the extracted method is clearly unparsable, because the variable color is not defined at the point of invocation of the new method. On the other hand, in the second approach, the additional generated #if directives can not be removed at all. The decision which of the two approaches should be implemented in

our tool was rather a question of taste. The second solution is safe, however, we do not like it, because of explosion of new generated #if conditionals. Possible reduction of new conditionals would be a hard task, taking into account how difficult the static analysis of cpp conditionals is [16]. Moreover, in majority of practical cases, user can always get sufficiently good extraction with the first approach. In cases when the first approach fails it is producing syntax error and, hence, there is no danger of introducing an unwanted bug to the runtime. For all those reasons, we have adopted the first approach in our implementation. Xrefactory generates non-guarded declarations of variables and it warns the user in cases when a variable does not occur in all preprocessor passes.

8 Conclusion

In this paper we have presented an implementation of a C++ refactoring browser. In order to deal with preprocessor, it computes refactorings in terms of source editing commands instead of AST transformations. Those editing commands are backmapped from preprocessed code to original unpreprocessed code and then applied. Conditionals are handled by multiple preprocessor passes and multiple parsings of source code. The final refactoring is combined from all parsings. This ad-hoc approach works well even in very complex circumstances and it allows to use a standard C++ parser taken from a compiler. In our implementation, we are using a professional compiler front-end produced by EDG company. The whole implementation consists of around 50 000 lines of code plus around 350 000 lines in the C++ parser. The implementation is available for download at the address *http://www.xref-tech.com/xrefactory*.

References

1. Eclipse. *http://www.eclipse.org*.
2. Ref++. *http://www.refpp.com*.
3. Visual slickedit. *http://www.slickedit.com*.
4. Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings of WCRE 2001: Working Conference on Reverse Engineering*, Stuttgart, Germany, 2001. IEEE Computer Society Press.
5. Kent Beck. *Extreme Programming explained*. Reading, MA, Addison Wesley Longman, Inc., 107108., 2000.
6. Anthony Cox and Charles Clarke. Relocating xml elements from preprocessed to unprocessed code. In *Proceedings of IWPC 2002: International Workshop on Program Comprehension*, Paris, France, 2002.
7. D.G.Waddington and B.Yao. High fidelity c++ code transformation. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005), Edinburgh University, UK*, 2005.
8. R. Fanta and V. Rajlich. Reengineering an object oriented code. In *Procceedings of IEEE International Conference On Software Maintenance*, pages 238–246, 1999.
9. Martin Fowler, (with contributions by K. Beck, J. Brant, W. Opdyke, and D. Roberts). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

11. A. Garrido and R. Johnson. Analyzing multiple configurations of a c program. In *Procceedings of IEEE International Conference On Software Maintenance, Budapest, Hungary*, 2005.
12. Alejandra Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 2005.
13. Alejandra Garrido and Ralph Johnson. Refactoring c with conditional compilation. In *18th IEEE International Conference on Automated Software Engineering*, Montreal, Canada, 2003.
14. Richard Garzaniti, Jim Huangs, and Chet Hendrickson. Everything i need to know i learned from the chrysler payroll project. In *Conference Addendum to the Proceedings of OOPSLA'97*, 1997.
15. Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–25, July 1988.
16. M. Latendresse. Fast symbolic evaluation of c/c++ preprocessing using conditional values. In *Procceedings of the seventh European Conference on Software Maintenance and Reengineering, Benevento, Italy*, pages 170–182, 2003.
17. Bill McCloskey and Eric Brewer. Astec: A new approach to c refactoring. *ACM SIGSOFT Software Engineering Notes*, 30(5), Sep 2005.
18. P. E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *International Conference on Compiler Construction, Varsovie, Pologne*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76, 2003.
19. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.
20. Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.
21. Don Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1999.
22. L. Tokuda and D. Batory. Evolving object-oriented architectures with refactorings. In *Conf. on Automated Software Engineering, Orlando, Florida*, 1999.
23. Mark van den Brand, Paul Klint, and Chris Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 2(32):54–61, February 1997.
24. Laszlo Vidacs, Arpad Beszedes, and Rudolf Ferenc. Columbus schema for c/c++ preprocessing. In *8th European Conference on Software Maintenance and Reengineering, Tampere, Finland*, pages 75–84. IEEE Computer Society, 2004.
25. Marian Vittek. A refactoring browser with preprocessor. In *Procceedings of the seventh European Conference on Software Maintenance and Reengineering, Benevento, Italy*, pages 101–111. IEEE Computer Society Press, 2003.