

EVALUATING COMMENT-TO-AST ASSIGNMENT HEURISTICS FOR C++ PROGRAMS

TAMÁS CSÉRI AND ZOLTÁN PORKOLÁB

ABSTRACT. Comments are integral part of the source code of software. They preserve the intentions of the developers, document constraints and highlight implementation details. Good comments help us to understand the codebase and make maintenance easier. Most of the software tools ignore comments because they take no part in code generation. However, there are cases when comments should be taken into account: refactoring tools need to move code along with their comments and code comprehension tools need to show comments related to a given context. Since these tools are working on the abstract syntax tree (AST), comments should be assigned to the appropriate AST nodes.

Assigning comments to AST nodes is a non-straightforward task. Most methods use heuristics that place the comment to the proper AST node. This article improves existing heuristics. We identify corresponding AST nodes by distance and type. We also manage to contract consecutive connected comments. Macro-related comments are handled in a special way. We quantify the correctness of comment assignments and evaluate the different solutions on open source C++ projects comparing our method with existing tools. Our method may be useful for other programming languages with respective modifications.

1. INTRODUCTION

Programming languages offer comments, that allows the programmer to store arbitrary textual data at almost any point of the source code. Developers may use comments to store information that can not be expressed using the constructs of the programming language: intentions behind the code, pre-conditions or other constraints, and explanation of the implementation details.

Received by the editors: May 1, 2014.

2010 *Mathematics Subject Classification.* 68N15, 68N20.

1998 *CR Categories and Descriptors.* D.1.0 [**Software**]: Programming Techniques – *General*; D.3.3 [**Software**]: Programming Languages – *Language Constructs and Features* D.2.7 [**Software**]: Programming Languages – *Distribution, Maintenance, and Enhancement* – *Documentation*.

Key words and phrases. Source Code Comments, Abstract Syntax Tree.

Carefully crafted comments help future developers to understand the codebase and make maintenance easier.

Comments are not necessarily structured textual data that most software tools skip during the processing of source code because they take no part in code generation. However, there are tools that are required to deal with comments. Refactoring tools are used to change the source code to increase maintainability, without changing the observable behaviour of the program. This sometimes needs moving segments of the source code to different location. Good refactoring tools not only move the code, but also move referring comments along. Code comprehension tools help users to understand and maintain complex systems. As comments carry important supplementary information, code comprehension systems benefit of understanding where a comment belongs and display them for the users at the appropriate places [12].

The compiler processes the source code in several steps. First, the character sequence of the source code is transformed to tokens by the lexer. In the optional next step the preprocessor (in case of languages like C or C++) resolves includes, macros and other preprocessor directives. Then the parser component of the compiler converts the preprocessed token sequence into a tree that describes the structure of the code. This tree is called abstract syntax tree, or AST for short. The AST is the canonical representation of the source code. Most program analysis tools work on the AST of the program because they are interested in the structure of the code and not the formatting details (like indentation, spacing, etc.) [11]. In a later step the semantic analyzer decorates the AST with various semantical information.

Refactoring and code comprehension tools also work with the AST. To handle comments, these tools first must map the comments to the AST [14]. For example, using a comment-to-AST mapping, refactoring tools can move the AST nodes and the comments describing them together. As an example, Eclipse CDT, one of the most widespread tools for C++ coding with refactoring support [1], work in this way. Sommerlad et al. present the mapping used by Eclipse CDT in [19].

In this article we examine the comment-to-AST mapping problem in C++ from a code comprehension view: whereas refactoring tools usually need to find only a single anchor point for comments so that source code operations (e.g. moving) keep code and the comments referring to a code together, our goal is to find all relevant AST nodes for a given comment.

We will use the open source Clang compiler of the LLVM toolchain to analyze the C++ source code. We chose Clang because it has a modular well-documented architecture, and it can be used as a library to create tools. [18].

This paper is organized as follows: in Section 2 we present a brief overview about comments in general and in the C++ programming language. In Section 3 we classify the comments into different categories. In Section 4 we introduce our algorithm to map comments to AST nodes. In Section 5 we elaborate on some important implementation details. Finally, in Section 6 we evaluate our method. Our paper concludes in Section 7.

2. COMMENTS IN THE C++ LANGUAGE

2.1. Comment notations. In the C programming language [16] comments start with `/*` and end with `*/`. These comments are usually referred as *block comments* or C-style comments. Block comments are sometimes called multi-line comments, because they can spread over multiple lines. C99 [7] and C++ [20] adds another notation for comments: everything between `//` and the end of the line is also treated as comment. We will refer them as *line comments* throughout the article.

By the standard, block comments can not be nested, however some compiler extensions allow this. In the rest of the paper we suppose that comments are compliant to the standard. Block and line comments can contain each other, in this case the content is formed as the union of the contents of the two components.

Throughout the article we call block or line comments that have content before and/or after them in the same line *inline comments*.

Line comments and block comments are the two kind of comments in most programming languages, therefore some of our results are applicable to other languages as well. Some programming languages lack the one or the other.

Preprocessor conditional statements are sometimes also used for commenting: lines between `#if 0` and the following `#endif` are ignored by the compiler completely. Contrary to block comments, preprocessor conditional statements can be nested, this makes this construction ideal for commenting out large blocks of code. Other uses of this commenting technique is rare.

2.2. Comments during the compilation of C++ programs. From the compiler point of view, the C++ standard specifies that during the 3rd phase of the translation, where the source file is decomposed into preprocessing tokens, “each comment is replaced by one space character” [8]. This allows the compiler to skip any comment during this very early phase, usually in the lexer component. Newer compilers, however, tend to retain comments during compilation. This allows compiler writers to emit more precise error location information that contains not only a line number but a column number as well.

3. CATEGORIZATION OF COMMENTS

As a first step, we group the comments into different categories. We observed that the two main types of comments are local comments that describe neighbouring entities and separator comments that describe subsequent code until the next separator comment. Additionally, we also distinguish code comments, which contain source code, and the mandatory copyright comments, which hold legal information about the file.

3.1. Local comments. Local comments describe neighbouring entities of the source code. Most comments in the examined projects are local comments.

As it can be seen in Figure 1, local comments come in various forms. They may appear before or after the referred content, they may be block or line comments, they may stand either in the same line of the referred content or in an adjacent line. Also, they may refer to multiple statements right

<code>// local comment for</code>	(1-a)
<code>// the main function</code>	(1-b)
<code>int main() {</code>	
<code> //local comment for variable x1</code>	(2)
<code> int x1;</code>	
<code> /* still a local comment */</code>	(3)
<code> int x2;</code>	
<code> int y; //local comment for variable y</code>	(4)
<code> int i1 /* inline */; int i2;</code>	(5)
<code> int z1; int z2; //last letter in</code>	(6-a)
<code> //the alphabet</code>	(6-b)
<code> //swap with helper variable</code>	(7)
<code> int tmp = z2;</code>	
<code> int z2 = z1;</code>	
<code> /}</code>	

FIGURE 1. Examples for local comments. (1), (2) and (3) are preceding line and block comments on their own lines. (4) is post comment, (5) is an inline comment, (6) is a post comment for multiple statements, also spanning multiple lines, (7) is a group comment for the three statements below.

before or right after them. In this section we provide heuristics to describe the characteristics of a local comment.

Our first notice is that block and line comments can be handled the same way as they do not behave differently.

Our next observation is that consecutive line comments (or block comments that are used like line comments) should be handled as a single comment if

- they are located in adjacent lines
- they start in the same column
- they have only whitespace characters in between

In Figure 1 we treat comments both (1-a) and (1-b) and (6-a) and (6-b) as a single comment.

We call local comments standing on lines of their own group comments if an empty line or a line ending with a block opening symbol (`{`) precedes them. Group comments refer to every statement following it, until the end of the current semantic block or an empty line.

Local comments that stand alone in a line but are not group comments refer to all AST nodes of the next line.

Comments standing the end of a line refer to preceding AST nodes. In Figure 1 (4) and (6) fall into this category. If the line contains multiple AST nodes (e.g. statements), there is no easy way to determine whether the comment refers to the whole line or only the last AST node. We applied the following guess: if there is more than one whitespace between the comment and the last non-whitespace character then it refers to the whole line, otherwise it refers to the last AST node only.

Comments in the middle of the line refer either to the AST node before or after them. We examine the whitespace before and after the inline comment and assign to the AST node on the side that is closer.

3.2. Documentation comments. A special case of local comments are documentation comments. Documentation comments document functionality of classes, functions and other names entities. The primary purpose of documentation comments is to provide information about the external interface of a function and describe its functional and non-functional behaviour.

Documentation comments are marked using a special notation: they usually begin with `/**` or `///`. External tools like Doxygen [22] parse these comments to generate a cross-referenced documentation of a project.

Although Doxygen allows comments for an entity to be written far from the described entity, this feature is rarely used [2]. Usually documentation comments are placed right before the explained function or class declaration. As this is sometimes inconvenient for member variables with short name and

```

/**                                     (1)
 * Documentation for class X
 */
class X {
  int x2; ///< special syntax for postfix comments  (2)
};

/*! \class X
    Also documentation for class X                (3)
*/

```

FIGURE 2. Documentation comments in the source code.

comment, we may use a marked line comment (e.g. starting with ///<) after the entity it documents.

Doxygen comments are well studied and existing parsing solutions can be used to map the comment to the AST node it documents. The Doxygen tool itself is the most complete implementation, recognizing several widespread formats, like the standard notation of Java [9] and C# documentation comments [10]. The C++ language has no recommended documentation style but Java and C# style comments are widely used along with own the comment notation of the Qt Framework ([5]).

Because of the aforementioned explicit notations, placing documentation comments in the AST is a well-defined problem, the heuristics we used for local comments are generally not needed. Whether a documentation comments refer to the following AST node and/or the previous AST node is encoded in its syntax. Inline documentation comments are not supported.

As a recent addition, Clang also implements documentation comment parsing [15]. Their goal is to emit warnings during compilation if the documentation is not consistent with the documented entity. For example if in a documentation comment we refer to a function parameter that does not exist in the function itself we get a warning (if the `-Wdocumentation` compile flag is on).

3.3. Separator comments. The source code is often organized by hierarchies that the programming language cannot express. For example if we have a class, we may want to express grouping on the methods. One way of this grouping is using comments to separate groups of code. An example is shown in Figure 3. We call these comments separator comments.

Some comments are marked explicitly as separator comments, for example comments centered by dashes or little boxes with some content centered inside. (Examples are (1) and (4) in Figure 3). We recognized them by the excess

```

class point
{
    [...]
public:

    /*****/                (1)
    /*      Getters      */
    /*****/

    //non-const getters    (2)

    int get_x();
    int get_y();

    //constant getters    (3)

    int get_x() const;
    int get_y() const;

    /*****/                (4)
    /*      Setters      */
    /*****/

    void set_x(int x);
    void set_y(int y);
};

typedef XPtr *X;

```

FIGURE 3. Separator comments with multiple levels. (1) and (4) are section separator comments, (2) and (3) are simple separators. Note that the `typedef` in the last line is outside the class scope and therefore (4) does not refer to it.

repetition of a single character and call them section separator comments. It is uncommon, but section separators may have multiple levels: e.g. different boxes for different levels. The recognition of different levels of section separator comments are project-specific expansions of our model.

We call simple separator comments the comments we recognize as separators because of the surrounding empty lines around them. More precisely, we assume that a comment is a simple separator comment if it is followed by an empty line – because if the comment would refer to the next statement only there would be no reason for the empty line – and is not a section separator

comment nor a documentation comment. All simple separator comments have the same level, which is greater than the level of the section separators.

3.4. Code comments. People often comment out legacy or not yet functional code. Although it is often considered to be a bad practise, we can encounter it in almost all examined projects.

One approach for distinguishing code comments from regular comment types is to try to parse them as source code and if we succeed it was a code comment. However, examples show that this is a too strict restriction – code comments may contain invalid code – e.g. because the commented code was never working or it may depend on code in another code comment to be valid.

Therefore, most comment analyzer tools take the approach of defining heuristics when a code should be considered code comment. Our algorithm uses this approach. We use the following heuristics, that we designed for recognizing code that look like C++ language.

We consider a comment to be a code comment suspect if either

- a comment ends with a block opening or closing symbol or a semicolon
- a valid preprocessor directive if found either at the beginning in of a line comment or at the first column of a line of a multiline comment.

This yields a high number of false positives e.g. it finds comments ending with `\LaTeX` documentation or multiline comments accidentally breaking at a documented preprocessor directive.

To reduce false positives we use the fact that three consecutive identifiers are rare in C++ source¹ but quite common in written text, so code comment suspects containing three consecutive identifiers will be treated as regular comments.

Code comments does not give information about the surrounding AST nodes, but we know that the code should be at the location of the comment, therefore we assign them to the encapsulating AST node.

3.5. Copyright comments. Due to legal requirements, in most projects every file begins with a *copyright comment* that provides a brief description on what the file does, a short extract of the license with a pointer to the full one and some additional metadata [2]. These comments refer to the whole file.

¹Three consecutive identifiers are only possible to write in C++ code because of macro expansion. Considering possibly defined macros may yield even better results. Two consecutive identifiers are possible in any declarations, e.g. `Foo bar;`.

4. COMMENT-TO-AST MAPPING ALGORITHM

In this section we describe our algorithm that maps the comments to the relevant AST nodes using the comment categories described in the previous section.

4.1. Collect comments and group them together if needed. First we extract comments with location information from the source code. We also contract consecutive comments to a single comment using the algorithm in Section 3.1.

4.2. Analyze comments and measure the surrounding whitespaces. Next we analyze the comments. As the first step, if it contains three or more successive special characters (*, -) we mark it as a section separator comment.

Otherwise we check if it is a documentation comment or it matches our code comment heuristics described in Section 3.4.

Next we count whitespaces around the comment: we count all whitespaces before and after the comment. We count newline characters separately to know the number of preceding and following empty lines. We use these numbers to decide whether it refers to the preceding or the following nodes for a single node, the whole line or until the next section comment having the level greater or equal to the level of the current section comment.

- Case 1: The comment on its own line(s). If it has an empty line after, we treat it as simple separator comment. If it has no empty line after, but an empty line before, it is a local group comment. Otherwise it is a local comment that is expanded for all the AST nodes of the next line.
- Case 2: The comment is in the end of the line. It refers to the preceding AST node. Whether a single node or the whole line is described in Section 3.1
- Case 3: The comment is inline. It refers to the nearest single AST node in the same line.

4.3. Assign comments to the AST. Our final step is to assign the comments to the AST nodes. We use breadth-first search to traverse the AST of the program. To take the structure of the code into account, for each comment we only consider the children of the deepest AST node that completely encapsulates our comment. From these child nodes we select the ones preceding or succeeding our comment (depending on the result of the analysis described in Section 4.2). We filter the selected AST nodes based on the category of the comment.

- For section comments we keep all nodes until the next section comment.
- For group comments we keep all nodes until the next empty line.
- For comments referring to a whole line we keep all nodes located in the current line.
- For other comments we keep the nearest match only.

This algorithm yields the relevant AST nodes for each comment.

4.4. Comments for macros and preprocessor symbols. The AST of the program contains neither the preprocessor directives nor macros, so our algorithm can not associate comments with them.

For preprocessor directives we introduce virtual AST nodes in the correct enclosing context. This way these virtual AST nodes get matched when looking for an AST node for a nearby comment.

For macros our algorithm works almost correctly as the comments can be associated with the AST nodes generated by the macro expansion. The only problem is when the expansion of a macro yields two or more sibling AST nodes and the algorithm associates the comment with only one of them. Our algorithm can cut macro expansions in half when it keeps only the first node while filtering the possible result list. Therefore we modify our algorithm to include the whole expansion.

5. IMPLEMENTATION

We implemented our comment-to-AST mapping algorithm using the Clang library as our backend for analyzing C++ source code and a plugin created for our code comprehension tool as the frontend to visualize our results.

5.1. Prerequisites. Although it is a bad programming practise to mix tabs and spaces but as open source projects are edited by a number of authors and the inconsistency is hidden by modern editors, we often found both indentation characters used in files. To ensure that our contraction algorithm works correctly, one should convert tabs to spaces using the correct project-specific tab-space ratio. Unix-based systems contain the `expand` utility for this task [6].

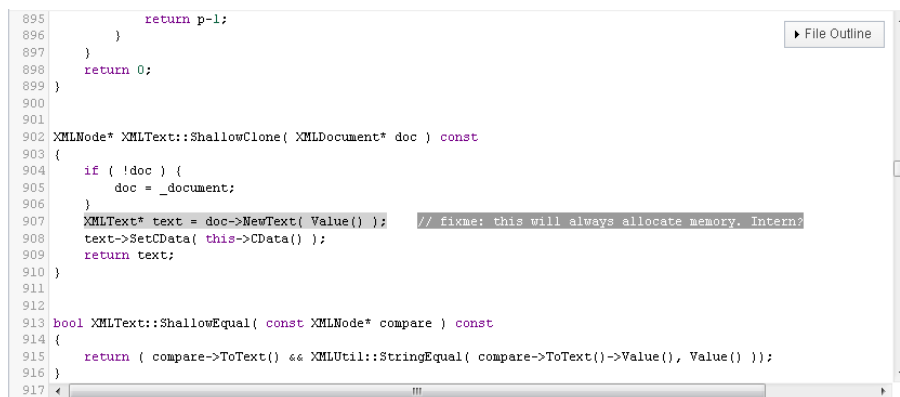
5.2. Backend. We built our backend based on Clang, the C/C++/Objective C compiler of the LLVM toolchain. We used Clang for extracting comments from the source code and we mapped the comments to the AST provided by Clang [17].

The Clang AST module has a `RawComment` structure that is suitable for representing comments. A vector of `RawComment` objects is stored along with the AST context of Clang in form of a `RawCommentList` object. Although its

name suggests otherwise, the `RawCommentList` object stores only `RawComment` objects of the documentation comments, as the Clang compiler currently parses the documentation comment only. Although there is a switch to parse all comments as documentation comments but `RawCommentList` also does merging of comments (which we want to control explicitly) and parsing the comments (which is not pointless for non-documentation comments). Therefore we patched the `RawCommentList` class of Clang to store every comment untouched as a `RawComment` object.

Using this complete list of `RawComment` objects as a starting point we implemented the algorithm described in Section 4. After our algorithm calculates the comment groups and their mapping to the AST we store the comments, comment groups and the AST mapping into a database.

5.3. Frontend. We visualized the results using the frontend of the code comprehension tool developed at Eötvös Loránd University in cooperation with Ericsson Hungary. We created a plugin that interactively highlights the whole comment group and the associated AST nodes upon clicking a comment. A screenshot of the tool highlighting a comment and the AST node the comment refers to can be seen in Figure 4.



```

895         return p-1;
896     }
897 }
898     return 0;
899 }
900
901
902 XMLNode* XMLText::ShallowClone( XMLDocument* doc ) const
903 {
904     if ( !doc ) {
905         doc = _document;
906     }
907     XMLText* text = doc->NewText( Value() ); // fixme: this will always allocate memory. Intern?
908     text->SetCDATA( this->CDATA() );
909     return text;
910 }
911
912
913 bool XMLText::ShallowEqual( const XMLNode* compare ) const
914 {
915     return ( compare->ToText() && XMLUtil::StringEqual( compare->ToText()->Value(), Value() ) );
916 }
917

```

FIGURE 4. A screenshot of the visualization interface. The “fixme” comment on the right is selected and the AST node it refers to is highlighted on the left.

6. EVALUATION

We evaluated our heuristics in the following way: we selected sample files from five open source projects, four implemented in C++ and one in pure C. A group of three C++ experts reviewed these projects with the help of the

frontend program presented in Section 5.3. In every comment occurrence the group of experts made a consensus decision whether the assignment suggested by our algorithm was correct.

The following five open source projects were evaluated:

TinyXML-2 [21] – TinyXML version 2 is a small XML parser library for C++. As it contains only three source files, we were able to analyze the whole project.

Xerces-C++ [13] – Xerces-C++ is another XML parser library for C++, with more complete XML support (e.g. schemas, XPath queries etc.).

Clang 3.4 [18] – We analyzed some files of the Clang compiler itself. We selected the files that extract comments from the C++ source.

GTest 1.7.0 [3] – Google Test is a unit testing library for C++. We analyzed its main header file and two additional source files.

libgit2 0.20 [4] – libgit2 is a new reimplementaion Git as a portable C library. We were interested if our methods apply to the very similar C programming language.

Project	File Name	LOC	DC	DCC	C	CC	Percentage
Tinyxml2	tinyxml2.h	2075	159	159	113	110	97%
Tinyxml2	tinyxml2.cpp	2187			66	63	95%
Tinyxml2	xmltest.cpp	1427	4	4	51	33	65%
Xerces-C	AbstractDOMParser.cpp	1778			79	66	84%
Xerces-C	DefaultHandler.hpp	807	49	49	4	3	80%
Xerces-C	ValueStackOf.hpp	156			14	13	93%
Clang	Stmt.h	2122	149	147	125	122	98%
Clang	Preprocessor.cpp	829	6	6	103	101	98%
Clang	RawCommentList.cpp	265	1	1	15	15	100%
GTest	gtest.h	2291			322	302	94%
GTest	gtest-filepath.cc	382			39	37	95%
GTest	gtest-messaging.h	250			31	30	97%
libgit2	blob.h	227	14	14	1	1	100%
libgit2	commit.c	350			9	8	89%
libgit2	tree.c	952	1		28	28	100%

TABLE 1. Evaluation results. Legend: LOC – lines of code, DC – number of documentation comments, DCC – number of correctly assigned documentation comments, C – number of non-documentation comments, CC – number of correctly assigned non-documentation comments, Percentage – percentage of correctly assigned non-documentation comments

Table 1 shows the results of our evaluation. Our comment-to-AST matching algorithm matches most of the comments to the referred AST nodes properly in open source programs. We found that each comment category defined in Section 3 appears in the inspected source code. In the examined Clang files we could see that our multi-level section comment recognition works correctly as well (e.g. between lines 1420 and 1509 of `Stmt.h`).

Most of the errors were due to the fact that local group comments standing at the beginning of a block sometimes refer to the whole block. This was also the cause of the high error rate in `xmltest.cpp`: each test case had its own block, and the summary of the test case was the first comment in this block. While these comments were referring to the whole test case, our algorithm categorized these comments as local group comments and assigned only the AST nodes until the first empty line, not the whole block of the test case. Assigning local group comments at the beginning of a block to the whole block would not work either: we found even more cases where the comment really referred to the first group only and not the whole block. Moreover, local group comments sometimes refer to the next line only, especially in member variable declaration lists.

Another common source of the mistakes (especially in Xerces-C++ and GTest) was that we could not correctly determine where a code section started by a section comment terminates. People often use section comments that refer only to some of the following AST nodes. A human reader can automatically “close” those comments upon seeing unrelated code, but the computer only sees the beginning section comment, and as there is no notation for closing a section comments, our algorithm assigns the section comment to unrelated

```
(Stmt.h - member functions of the CompoundStmt class)
00627 // Iterators
00628 child_range children() {
00629     return child_range(Body, Body + CompoundStmtBits.NumStmts);
00630 }
00631
00632 const_child_range children() const {
00633     return child_range(Body, Body + CompoundStmtBits.NumStmts);
00634 }
```

FIGURE 5. Incorrect placement of comment due to the lack of semantic information of its content. The human reader recognizes the plural form and that both methods return iterators and assigns the comment for both methods.

parts of the AST as well. We even found simple section comments being used in a nested way.

There were various other reasons for the incorrect assignments as well. For example, Figure 5 shows a comment that our algorithm interprets as a local comment for a function, but it clearly refers to both functions. We found similar erroneous usage of documentation comments as well.

The errors described above can only be recognized by understanding the meaning of the comment. Our comment-to-AST mapping uses only the placement information of the comment which allowed us to create a well-defined deterministic algorithm. However, due to the unstructured nature of these comments, sometimes more advanced techniques are needed for proper comment handling. By analyzing the contents of the comments we might be able to improve the percentage of correctly mapped comments.

Extending coding style guides with exact comment positioning and formatting instructions and adherence to these stricter commenting conventions would help future programs to be processed more accurately. The programmers might adopt to the stricter standards if they see that they get better refactoring results and code comprehension in return.

Finally, we must also note, that the coding guidelines of open source programs are very strict and low quality code is often rejected during code review. In companies without strict authoring policies we expect lower comment match percentage.

7. CONCLUSION

Comments preserve the intentions of the developers, document constraints and highlight implementation details and therefore they are a crucial key for understanding programs. While many software tools ignore comments, there are use cases where the proper assignment of comments to the abstract syntax tree is essential. In this paper first we categorized comments then we provided an algorithm based on heuristics that maps each kind of comment to the proper AST node(s) they refer to. We also addressed problems specific to the C++ language like comments for macros and other preprocessor-related elements.

To validate our algorithm, we created an implementation to assign comments to AST nodes using the Clang open source C++ compiler and a graphical frontend to show the resulted mapping. We evaluated our algorithm on different open source C++ codebases. We experienced that our heuristics were mostly correct. By analyzing our mistakes we noticed that positional information is not enough to correctly assign the comments to the AST nodes. Our further research will target these situations analyzing semantic connections between comment text and program elements.

REFERENCES

- [1] Eclipse CDT (C/C++ development tooling).
<http://www.eclipse.org/cdt/>.
- [2] Google C++ style guide.
<http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>.
- [3] Google C++ testing framework.
<https://code.google.com/p/googletest>.
- [4] libgit2 – a portable, pure C implementation of the Git core methods.
<http://libgit2.github.com>.
- [5] QDoc manual.
<http://doc-snapshot.qt-project.org/qdoc/qdoc-index.html>.
- [6] `expand`: convert tabs to spaces. In *Commands & Utilities Reference, The Single UNIX Specification, Issue 7*. The Open Group.
- [7] *ISO/IEC 9899:1999 – Programming languages – C*. ISO, Geneva, Switzerland, 1999.
- [8] *ISO/IEC 14882:2011 – Programming Language C++*. ISO, Geneva, Switzerland, 2003.
- [9] Javadoc 5.0 tool.
<http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/index.html>, 2010.
- [10] C# language specification 5.0.
<http://www.microsoft.com/en-us/download/details.aspx?id=7029>, 2012.
- [11] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.
- [12] T. Cséri, Z. Szügyi, and Z. Porkoláb. Rule-based assignment of comments to AST nodes in C++ programs. In *Proceedings of the Fifth Balkan Conference in Informatics, BCI '12*, pages 291–294, New York, NY, USA, 2012. ACM.
- [13] A. S. Foundation. Xerces-c++ xml parser. <http://xerces.apache.org/xerces-c/>.
- [14] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [15] D. Gribenko. Parsing documentation comments in Clang. LLVM Developers' Meeting, 2012.
- [16] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.
- [17] M. Klimek. The Clang AST – a tutorial. LLVM Developers' Meeting, 2012.
- [18] C. Lattner et al. Clang: a C language family frontend for LLVM.
<http://clang.llvm.org/>.
- [19] P. Sommerlad, G. Zraggen, T. Corbat, and L. Felber. Retaining comments when refactoring code. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, OOPSLA Companion '08*, pages 653–662, New York, NY, USA, 2008. ACM.
- [20] B. Stroustrup. *The C++ Programming Language: Special Edition*. Addison-Wesley, Reading, MA, 2000.
- [21] L. Thomason. TinyXML-2.
<http://www.grinninglizard.com/tinyxml2/>.
- [22] D. van Heesch. Doxygen – source code documentation generator tool.
<http://www.stack.nl/~dimitri/doxygen/>.

The URLs were retrieved at: 1 May 2014.

DEPARTMENT OF PROGRAMMING LANGUAGES AND COMPILERS, FACULTY OF INFORMATICS, EÖTVÖS LORÁND UNIVERSITY, PÁZMÁNY PÉTER SÉTÁNY 1/C, 1117 BUDAPEST, HUNGARY

E-mail address: {cseri,gsd}@caesar.elte.hu