Annales Mathematicae et Informaticae
44 (2015) pp. 23–33
http://ami.ektf.hu

Component visualization methods for large legacy software in C/C++

Máté Cserép[a], Dániel Krupp[b]

[a]Eötvös Loránd University
mcserep@caesar.elte.hu

[b]Ericsson Hungary
daniel.krupp@ericsson.com

Submitted August 28, 2014 — Accepted February 24, 2015

Abstract

Software development in C and C++ is widely used in the various industries including Information Technology, Telecommunication and Transportation since the 80-ies. Over this four decade, companies have built up a huge software legacy. In many cases these programs, implementing complex features (such as OS kernels, databases) become inherently complicated and consist of millions lines of code. During the many years long development, not only the size of the software increases, but a large number (i.e. hundreds) of programmers get involved. Mainly due to these two factors the maintenance of software becomes more and more time consuming and costly.

To attack the above mentioned complexity issue, companies apply various source code cross-referencers to help in the navigation and visualization of the legacy code. In this article we present a visualization methodology that helps programmers to understand the functional dependencies of artifacts in the C++ code in the form similar to UML component diagrams. Our novel graph representation reveals relations between binaries, C/C++ implementation files and headers. Our technique is non-intrusive. It does not require any modification of the source code or any additional documentation markup. It solely relies on the compiler generated Abstract Syntax Tree and the build information to analyze the legacy software.

Keywords: code comprehension, software maintenance, static analysis, component visualization, graph representation, functional dependency

MSC: 68N99

1. Introduction

One of the main task of code comprehension software tools is to provide navigation and visualization views for the reusable elements of the source code, because humans are better at deducing information from graphical images [2, 7]. We can identify reusable software elements in C/C++ language on different abstraction levels of modularity. At a finer granularity, functions provide reusable implementation of a specific behavior, while on a higher scale (in C++) classes defines the next level, where a programmer can collect related functions and data that belong to the same subject-matter. At the file level, header files group related functions, variables, type declarations and classes (in C++ only) into a semantic unit.

State of the art software comprehension and documentation tools implement various visualization methods for all of these modularization layers. For example, on the function level call graph diagrams can show the relations between the caller and the called functions [9], while on the class level, one can visualize the containment, inheritance and usage relations by e.g. UML diagrams. On the file level, header inclusion diagrams help the developers in code comprehension [8].

However, our observations showed that the state of the art file level diagrams are not expressive enough to reveal some important dependency relationships among implementation and header files. In this paper, we describe a new visualization methodology that exposes the relations between implemented and used header files and the source file dependency chains of C/C++ software.

This paper is structured as follows. Section 2 consist of a brief overview of the state of the art literature with special focus on static software analysis. In Section 3 we describe the shortfalls of the visualization methods in the current software comprehension tools, then in Section 4 we present our novel views that can help C and C++ programmers in understanding legacy source code. Section 5 demonstrates our results by showing examples on real open-source projects, and finally in Section 6 we conclude the paper and set the directions for future work.

2. Background

Researchers have proposed several software visualization techniques and various taxonomies have been published over the past years. They address one or more of three main aspects (static, dynamic, and evolutional) of a software. The visualization of the static attributes focuses on displaying the software at a snapshot state, dealing only with the information that is valid for all possible executions of the software, assisting the comprehension of the architecture of the program. Conversely, the visualization of the dynamic aspects shows information about a particular execution of the software, therefore helps to understand the behavior of the program. Finally, the visualization of the evolution – of the static aspects – of a software handles the notion of time, visualizing the alternations of these attributes through the lifetime of the software development. For a comprehensive summary of the current state of the art see the work of Caserta et al.[3]

The static analysis of a software can be executed on different levels of granularity based on the level of abstraction. Above a basic source code level, a middle – package, class or method – level, and an even higher architecture level exists. In each category a concrete visualization technique can focus on various different aspects. A summary of categorization is shown on Table 1, classifying some of the most known and applied, as well as a few interesting new visualization techniques.

<table>
<tr><th>Kind</th><th>Level</th><th>Focus</th><th>Techniques</th></tr>
<tr><td rowspan="7">Time T Visualization</td><td>Line</td><td>Line properties</td><td>Seesoft</td></tr>
<tr><td>Class</td><td>Functioning, Metrics</td><td>Class BluePrint</td></tr>
<tr><td rowspan="5">Architecture</td><td>Organization</td><td>Treemap</td></tr>
<tr><td rowspan="4">Relationship</td><td>Dependency Structure Matrix</td></tr>
<tr><td>UML Diagrams</td></tr>
<tr><td>Node-link Diagrams</td></tr>
<tr><td>3D Clustered Graphs</td></tr>
<tr><td colspan="4">Visualizing Evolution</td></tr>
</table>

Table 1: Categorization of visualization tools

This article focuses on assisting the code comprehension through visualizing the relationships between architectural components of a software. The relevant category not only contains various prevalent and continuously improved visualizing techniques like the *UML diagrams* [4], but also recently researched, experimental diagrams like the three dimensional clustered graphs [1]. This technique aims to visualize large software in an integral unit, by generating graphs in a 3D space and grouping remote vertices and classes into clusters. The visibility of the inner content of a cluster depends dynamically on the viewpoint and focus of the user who can traverse the whole graph.

Our novel solution uses the classical node-link diagram in two dimensional space for visualization, which was formerly used at lower abstraction levels primarily.

3. Problems of visualization

Modularity on the file level of a software implementation in C/C++ is expressed by separating interfaces and definition to header and implementation (source) files. Interfaces typically contain macro and type definitions, function and member declarations, or constant definitions; while implementation files usually contain the definition of the functions declared in the headers. This separation allows the programmers to define reusable components in the form of – static or dynamic – libraries. Using this technique, the user of a library does not need to have information about the implementation details in order to use its provided services.

Separation of these concerns is enforced by the C/C++ preprocessor, compiler and linker infrastructure. When a library is to be used, its header file should be *included* (through the `#include` preprocessor directive) by the client implementation or the header files. Implementation files should almost never[1] be included in a project where the specification and implementation layers are properly separated. Unfortunately naming conventions of the header and implementation files in C/C++ are not constrained (like calss and file-naming in Java). Thus, based on a name of a file, it is not possible to find out the location, where the methods of a class are declared or implemented. Furthermore, the implementation of the class members declared in a header file can be scattered through many implementation files that makes the analysis even more difficult.

When a programmer would like to comprehend the architecture of a software, the used and provided (implemented) interface of a library component or the implementers of a specific interface should be possible to be fetched.

Problem 3.1. As an example let us analyze the commonly presented header inclusion graph of a fileset in Figure 1. We assume that `lib.h` is an interface of a software library and that there are many users of this component, thus several files includes this header. If the programmer would like to comprehend where the functions declared in the header are implemented, the header inclusion graph is not helpful, since it does not unveil which C/C++ files are only using, and which are implementing the `lib.h` interface.

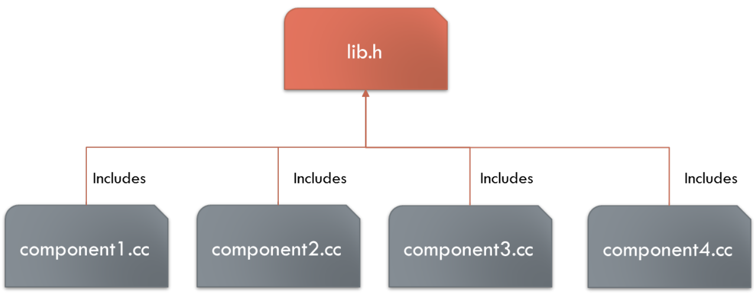

Figure 1: Implementation decision problem between component(s) and an interface.

As a solution we propose a so-called *Interface diagram* that is similar to the well-known header inclusion graph, but refines the include relation into *uses* and *provides* relationships. For this purpose we defined that a C/C++ file provides a header file when it contains its implementation, while it only uses it if the mentioned file refers to at least one symbol in the header, but does not implement any of them. A proper and precisely defined description of this diagram is given in Section 4.2.

[1]A few exceptions may exist, i.e. in some rare cases of template usage.

4. Definition of relationships and diagrams

In this section first we introduce the commonly used basic terms of relationships defined between the C/C++ source files and the binary objects (see Figure 2), then present our more complex relationship definitions to describe the connections between the various kind of files in a software project at a higher abstraction level.

4.1. Preliminaries

Definition 4.1 (Relations between source files)**.** At the level of the *abstract syntax tree* [6], the main artifacts of a C/C++ source code are the user defined symbols[2], which can be declared, defined or referred/used by either the source files (`.c`/`.cc`) or the header files (`.h`/`.hh`). A C/C++ symbol might have multiple declarations and references, but can be defined only once in a semantically correct source code. To enforce the separation of the specification and implementation layer, header files should mainly consist of declarations, whose definitions are in the appropriate source files.[3] From our perspective only those C/C++ symbols are important, which are declared in a header file and are defined or referred by a source file.

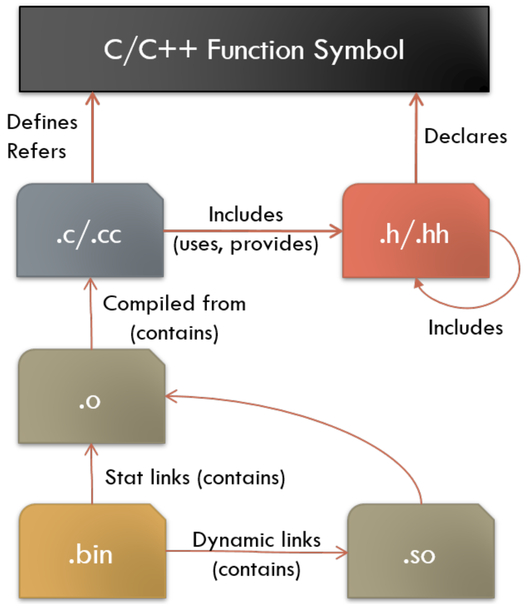

Figure 2: Relations between compilation artifacts.

[2]From our viewpoint only the function (and macro) symbols and their declaration, definition and usage are significant, although a similar classification for other symbol types can be established without difficulties.

[3]In some cases, headers may contain definition and source file may also consist forward declarations as an exception.

Definition 4.2 (Relations between binaries)**.** The source files of a project are *compiled* into object files, which are then *statically linked* into archive files (.lib/.a), shared objects or executable binaries. Shared objects are *linked dynamically* into the executables at runtime. To extract this information and visualize the relationship of binaries together with the relations declared between the C/C++ files, the analysis of the *compilation procedure* of the project is required beside the static analysis of the source code.

For the purpose of the presented visualization views in this paper the different kind of binary relationships is irrelevant, therefore they will be collectively referred as the *contains* relation henceforward (see Figure 2).

4.2. Extended classification

The basic include relationship among the implementation and header C/C++ files have already been introduced in Section 4.1, however in order to solve the problem raised in Section 3, the definitions of the proposed uses and provides relations have to be separated.

Definition 4.3 (Provides relationship from implementation c to header h)**.** We say that in a fileset a implementation file c *provides* the interface specified by the header file h, when c directly includes h and a common symbol s exists, for which h contains the declaration, while c consists the definition of it.

Definition 4.4 (Uses relationship from implementation c to header h)**.** Similarly to the previous provides relationship definition, we state that in a fileset an implementation file c *uses* the interface specified by the header file h, when c directly includes, but does not provides h and a common symbol s exists, which c refers and h contains the declaration of it.

Definition 4.5 (**Interface graph** (diagram) of implementation file c)**.** Let us define a graph with the set of nodes N and set of edges E. Let P be the set of header files which are *provided* by c and U the set of header files used by c, and B the set of binary files which *contain* c. We define that N consists of c, the elements of P, U, and B. E consists the corresponding edges to represent the relationships between the nodes in N.

Figure 2 shows the illustration for the above mentioned definitions. Based on the idea of the *Interface diagram* defined in Definition 4.5, which shows the immediate *provides*, *uses* and *contains* relations of the examined file, we defined the following more complex file-based views.

The nodes of these diagrams are the files themselves and the edges represent the relationships between them. A labeled, directed edge is drawn between two nodes only if the corresponding files are in either *provides*, *uses* or *contains* relationship. The label of the edges are the type of their relationship and they have the same direction as the relation they represent.

Definition 4.6 (**Used components graph** (diagram) of source c)**.** Let us define a graph with the set of nodes N and set of edges E, and let S be the set of implementation files which *provides* an interface directly or indirectly *used by c*. We define that N consists of c, the elements of S and the files along the path from c to the elements of S. Binaries containing any implementation file in S are also included in N. E consists the corresponding edges to represent the relationships between the nodes in N.

Intuitively we can say if source t is a used component of c, then c is using some functionality defined in t.

Definition 4.7 (**User components graph** (diagram) of source c)**.** Let us define the graph with the set of nodes N and set of edges E, and similarly to the previous definition, let S be the set of implementation files which directly or indirectly *uses* the interface(s) *provided by c*. We define that N consists of c, the elements of S and the files along the path from c to the elements of S. Binaries containing any implementation file in S are also included in N. E consists the corresponding edges to represent the relationships between the nodes in N.

Intuitively we can say if source t is a user component of c, then c is providing some functionality used by t.

5. Experimental results

In order to implement the views defined in Section 4, we created a diagram visualizing tool was created as part of a larger code comprehension supporting project – named *CodeCompass*. The software is developed in cooperation at Eötvös Loránd University and Ericcson Hungary. The tool provides an interactive graph layout interface, where the users are capable of requesting more information about the nodes representing files and can also easily navigate between them, switching the perspective of the view they are analyzing.

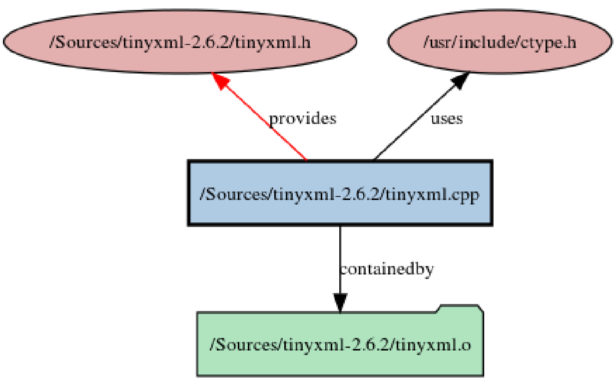

Figure 3: Interface diagram of `tinyxml.cpp`.

For demonstration purposes in this paper, the open-source *TinyXML* parser project[10] was selected. In this section altogether three examples for the use of our tool is shown and information retrievable from them is examined.

Example 5.1. Figure 3 displays an *Interface diagram*, showing the immediate relations of a selected file with other files in the software. As the image shows, the C++ implementation file in the middle (`tinyxml.cpp`) includes two header files, but the special connection of implementation (provides) is distinguished from the mere uses relation. This diagram not only presents the connections between C++ source and header files, but also displays in which object file the focused implementation file was compiled into through the compilation process of the project.

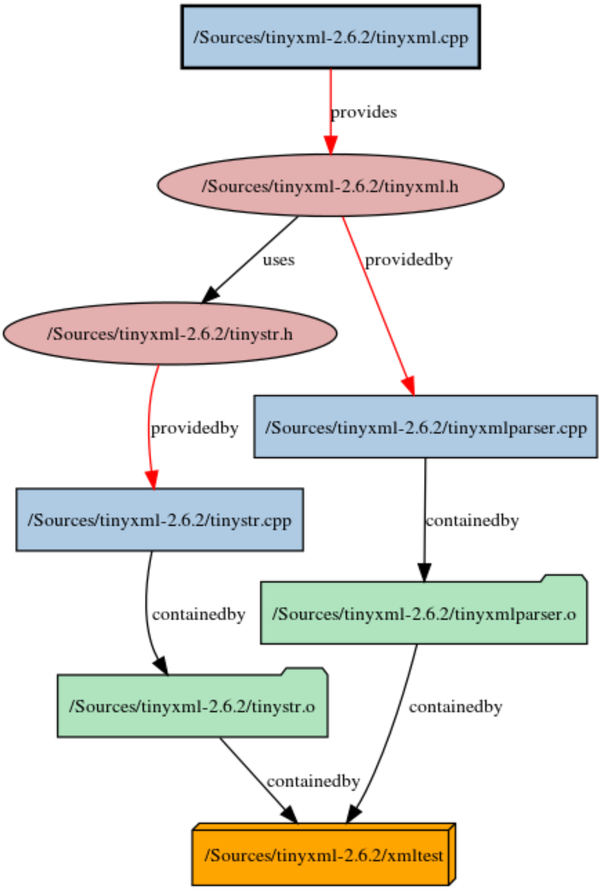

Figure 4: Used components by `tinyxml.cpp`.

Example 5.2. Figure 4 presents the *Used components diagram* of the implementation file `tinyxml.cpp` at the top. The goal of this visualization is to determine which other files and compilation units the selected file depends on. As it is depicted in the figure, the interface specification for the `tinyxml.cpp` implementation file is located in the `tinyxml.h` header. This header file on the one part is provided by the `tinyxmlparser.cpp`, and on the other hand uses the `tinystr.h`. The latter header file is provided by the `tinystr.cpp` source. Hence the implication can be stated that the original `tinyxml.cpp` indirectly uses and depends on the `tinyxmlparser.cpp` and the `tinystr.cpp` file.

Example 5.3. Parallel to Figure 4, the following example deduces the compilation artifacts depending on the same selected source `tinyxml.cpp`. The *User components diagram* displays (see Figure 5) that this implementation file implements an interface contained by the `tinyxml.h` header. This header is used or provided by three sources (`tinyxmlparser.cpp`, `xmltest.cpp` and `tinyxmlerror.cpp`), therefore they are the users of `tinyxml.cpp`.

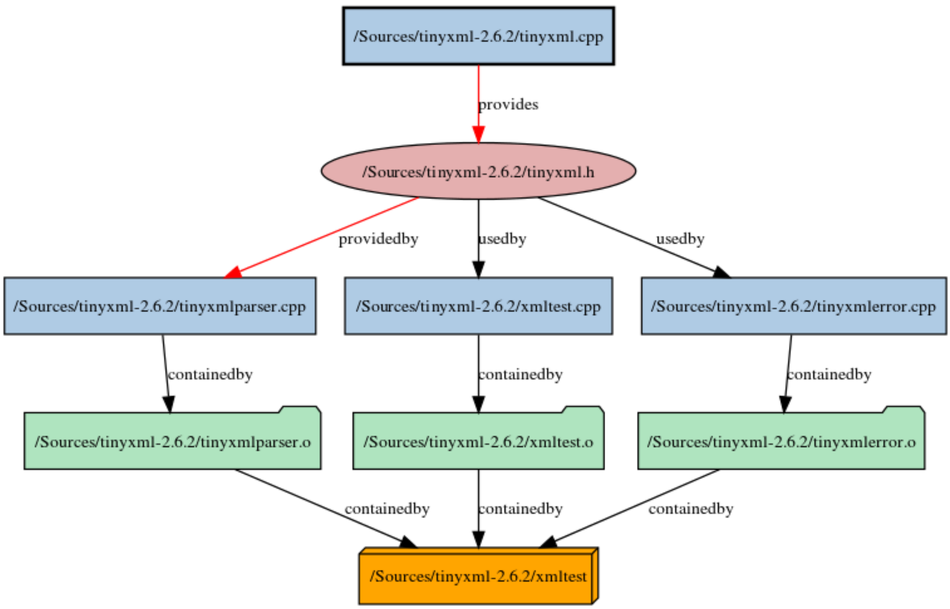

Figure 5: User components of `tinyxml.cpp`.

6. Conclusions and future work

In large legacy software projects a huge codebase can easily be built up because of the extended development time, while fluctuation among programmers can also often be a significant problem. Code comprehension support addresses these questions through assisting – both experienced and newcomer – developers with visualization views to better understand the source code. In this paper we discussed what kind of file-level views are missing from the current code comprehension tools, regarding the relationships between different type of compilation artifacts. We defined our novel graph view as a solution to this problem, and demonstrated the practical use of our technique through examples on an open-source C++ project. The new visualization techniques were found helpful and applicable for legacy software in supporting code comprehension.

Above the file-level granularity, a higher level of modularity can also be defined, considering that related files can form the interface of a reusable binary component and are often grouped together physically (i.e. contained in a directory) or virtually (e.g. using packages or namespaces). Future development will generalize and expand the file-based dependency relationship definitions introduced in this paper to be applicable for modules containing multiple files.

Further work will also include the examination of how the information retrieved by our definition rules can be used in the field of architecture compliance checking. Software systems often impose constraints upon the architectural design and implementation of a system, for example on how components are logically grouped, layered and how they may interact with each other. In order to keep the maintainability of a software system through a long development time with a large programmer team, it bears extreme importance that the design and implementation are compliant to its intended software architecture. Due to the complexity of large software systems, guaranteeing the compliance by manual checking is almost impossible, hence automated support is required, which is still not a completely solved issue nowadays [5].

References

[1] Balzer, M., Deussen, O., Level-of-detail visualization of clustered graph layouts, *Proceedings of the 6th International Asia-Pacific Symposium on Visualization*, (2007), 33–140.

[2] Biederman, I., Recognition-by-components: a theory of human image understanding, *Psychological review*, Vol. 94 (1987), 115–147.

[3] Caserta, P., Zendra, O., Visualization of the static aspects of software: a survey, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 17 (2011), 913–933.

[4] Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., Mutzel, P., A new approach for visualizing UML class diagrams, *Proceedings of the 2003 ACM Symposium on Software Visualization*, (2003), 179–188.

[5] Pruijt, L., Koppe, C., Brinkkemper, S.,, On the accuracy of architecture compliance checking support: Accuracy of dependency analysis and violation reporting, *IEEE 21st International Conference on Program Comprehension*, (2013), 172–181.

[6] Salomaa, A., Formal Languages, *Academic Press Professional, Inc.*, (1987).

[7] Spence, I., Visual psychophysics of simple graphical elements, *Journal of Experimental Psychology: Human Perception and Performance*, Vol. 16 (1990), 683–692.

[8] Doxygen Tool: `http://www.stack.nl/~dimitri/doxygen/`.

[9] Understand Source Code Analytics & Metrics, `http://www.scitools.com/`.

[10] TinyXML parser, `http://www.grinninglizard.com/tinyxml/`.