Supporting Code comprehension with CodeCompass

Zoltán Porkoláb*,†

[1]ELTE Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers, Pázmány Péter sétány 1/C, 1117 Budapest, Hungary

Abstract

During software development, whether we are developing new functionality, fixing a bug or modifying an existing operation, it is paramount to foresee all the consequences of the planned changes. This cannot be done without fully understanding the given software system. Unfortunately, in the case of long-lived, large-scale software systems, it is common for the original intentions and decision-making criteria to fade into the past over the decades due to the fluctuation in the development team. Documentation is often incomplete, inaccurate – if it exists at all. All this means that the only reliable information we have about the software is the source code itself. Understanding such industrial-scale systems, often consisting of many millions of source lines, is an extraordinary challenge even for experienced developers. There is a natural need to support this comprehension process with tools. One such tool is CodeCompass, an open source, extensible code comprehension framework that uses static analysis to help developers better understand industrial-scale code. CodeCompass supports multiple programming languages through plug-ins and uses a variety of textual and graphical visual elements to enhance comprehension.

Keywords

Code comprehension, Static analysis, CodeCompass,

1. Introduction

The maintenance of large, long-lived software systems is notoriously problematic and can take up a large portion of the total cost of the software over its entire life cycle. Such systems, which have been developed, used, and maintained for decades, often involve hundreds of developers [1, 2], who are replaced in time. With every developer who leaves the project, a part of the knowledge about the system is lost, and for every new entrant, acquiring the necessary knowledge is a serious investment of time and effort. However, neither effective bug fixing nor reliable new development can be achieved without the IT specialists working on the task being able to confidently understand the specific details of the system and its relationships with other modules.

For these reasons, today's average software developer usually spends much more time understanding existing software than writing new code. Individual research shows different numbers, but there is general agreement that code understanding tasks today reach up to 80% of developers' time. The situation is particularly critical when a completely new team takes over the development of a software, and it is also observed that younger, less experienced developers need more time to understand the system.

It is natural that software support arises in the case of a task of this scale. In recent decades, integrated development environments (so-called IDEs) have been gradually supplemented with intelligent support elements. Such elements are code completion, which offers a meaningful continuation of the source code based on the code environment; functionality that allows closing and opening individual code sections; integration of version control systems; and the incremental compilation of the project. At the same time, the primary use of integrated development environments is writing new source code and

SQAMIA 2026: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 16–18, 2026, Nis, Serbia

*Corresponding author.

gsd@inf.elte.hu (Z. Porkoláb)

https://gsd.web.elte.hu/ (Z. Porkoláb)

0000-0001-6819-0224 (Z. Porkoláb)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-19

CEUR-WS.org/Vol-4336/short1.pdf

their tool support mainly supports this goal. However, writing new code is significantly different from understanding existing code base.

When writing new source code, we typically spend a lot of time at a given level of abstraction. In the case of C and C++ programs, we design the interface and create one or two header files based on it. After this, or perhaps in parallel, we start working on the implementation details, which we write into a source file. In the case of test-driven development, this can be supplemented by writing the actual unit-tests, which can mean another source file. Typically, we have no more than 4-5 files open at a time, and we switch between them while editing the files. The split of windows in current development tools is perfectly suited to this approach, where we find each file in individual windows, or possibly in separate tabs that can be opened. We call this common visual approach the "bento-box" style, after the name of the Japanese lunch box. The disadvantage of the bento-box is that you cannot see more than 3-4 important related elements on the screen at the same time, we do not see the relationships between the files, and we often have to spend time navigating in individual windows to find the elements that interest us. However, this format is perfectly suited to developing new code, where we work sequentially, spending more time editing the same few files.

However, the process of understanding source code differs significantly from that scheme. When learning about an unknown or only partially known piece of software, we have to open many files, examine the classes and functions they defines, and discover the relationships between them. Meanwhile, the level of abstraction applied is constantly changing: once we examine the implementation of a function, then we can look at the class containing it, and from there we have to assess the class relationships. Often, we have to display 10-15 essential program elements (often located in different files) and their relationships at the same time. This does not work with bento-box visualization. The other significant difference is that while writing new code is typically textual, it is possible and even worthwhile to use graphical representation for understanding, as it provides greater bandwidth during the understanding process than textual representation.

Based on these differences, tools specifically supporting code understanding were created. These tools provide developers with fast searches, precise code navigation, and a comprehensive graphical notation system. One such tool is CodeCompass, an open source, free-to-use framework [3]. CodeCompass is a plug-in-based, static analysis system that uses the full range of information available about the software; in addition to the source code, e.g., compilation and editing commands or git version control data. The information obtained in this way is stored in a database and then transmitted to the user using a web browser or REST API. CodeCompass always knows a snapshot of the software, and in case of changes, fast incremental parsing is provided.

2. The process of code comprehension

Code comprehension has long been an important and well-studied part of the software development process. Although every piece of software is different, and every programmer develops their own strategy for understanding large systems, we can still examine this process with scientific rigor. Among others, von Mayrhauser [4], Storey [5], and O'Brien [6] have collected, examined, and categorized models of comprehension.

As von Mayrhauser et al. describe [4], there are several common elements in different strategies that are independent of the system to be understood. These are partly static, such as plans that developers follow, program elements that are attempted to be understood independently of other elements, hypotheses that are formulated about the code and are either confirmed or rejected. On the other hand, there are also dynamic elements, such as the search for connections with other parts of the program. However, in most cases, developers either proceed from the larger structure of the program to smaller details (top-down method) or, after understanding smaller program elements, try to synthesize knowledge in order to understand the whole system (bottom-up method).

According to Brooks' model, developers proceed in a top-down manner, making micro-hypotheses about the code based on their domain knowledge and either accepting and refining, or rejecting them

and creating alternative hypotheses based on the details they learn [7].

Soloway, Adelson, and Ehrlich describe another top-down method in their article [8]. According to their model, during understanding, developers set hierarchical understanding goals, starting with expectations for the entire system. From there, they try to iteratively refine their ideas in response to the code they learn. The authors believe that top-down methods are primarily used by developers when they encounter known code patterns and program structures.

However, some have primarily analyzed bottom-up methods. Pennington [99] distinguishes two mental models: the program model and the situation model. The former is based on the control structure of the program and is thus built from the bottom up. In the situational model, we consider the (partial) goals of the program and the data flow. Here too, we approach it from the details, but we take into account the domain-specific knowledge and build a hierarchical model.

Shneiderman and Mayer examined the direction of the comprehension process in terms of memory, as knowledge gained by processing program code and existing higher-level ideas moves from short-term memory to long-term memory [9]. Their model distinguishes between syntactic and semantic knowledge. While the former is based on, for example, recognizing elements of the programming language used, the latter extends from low-level practical parts of the code through program abstractions to higher-level goals of the software.

Levy [10] notes that the top-down comprehension process is usually used by developers when their goal is to understand the entire system architecture or the role of certain larger modules. The bottom-up approach is primarily used when fixing bugs or implementing a new feature.

An important concept in the practical application of code comprehension is the localization of functionality [11]. When we fix a bug or add new functionality to a system, we first need to find the part of the software that is the most relevant to the task. Sometimes this area can be identified directly: there exists a class, function, or some other similar program element that addresses it. Sometimes, however, we only find the right place implicitly, based on some similar abstraction or prerequisite [12].

Feature localization is not tied to any specific understanding model, but can rather be seen as the starting point for all models. In the first step, we try to find the code fragment that interests us, and from there (using a top-down or bottom-up method) we expand our knowledge of the system.

3. Tools supporting code comprehension

There are many software tools that directly or indirectly support code understanding. Some are developed for the purpose of code development, such as integrated development environments (IDEs), while others are specifically designed for code understanding.

Most integrated development environments, such as Visual Studio, IntelliJ, Netbeans, Eclipse, Code-Blocks, QtCreator, have some code navigation functionality. However, this is usually limited to jumping to the definition (GoTo definition) and finding all occurrences of a name (All references). Even in these systems, it is not uniform whether only a name search is performed in such cases or the semantically correct symbol is resolved. These options are primarily convenient when developing code, but are not necessarily suitable for localizing functionality (if, for example, we do not know the exact name of the element we are looking for) and displaying and browsing the results is not convenient. On the other hand, in the case of large projects, these tools are often unable to index the entire system, so the search results may be incomplete. However, it must be acknowledged that IDEs have undergone significant development and it is expected that more and more comprehension-supporting elements will be integrated into their toolkit.

Among the tools specifically created to support code comprehension, we must distinguish between tools that provide fast, convenient, but not necessarily accurate code navigation and software that performs deeper semantic checking.

The former group includes, for example, **OpenGrock** [13], which is a fast cross-reference system. These tools read the source code, perform some lexical and syntactic analysis, but do not perform deeper semantic checks, name or overload resolution. OpenGrock uses, for example, the ctags [14] library,

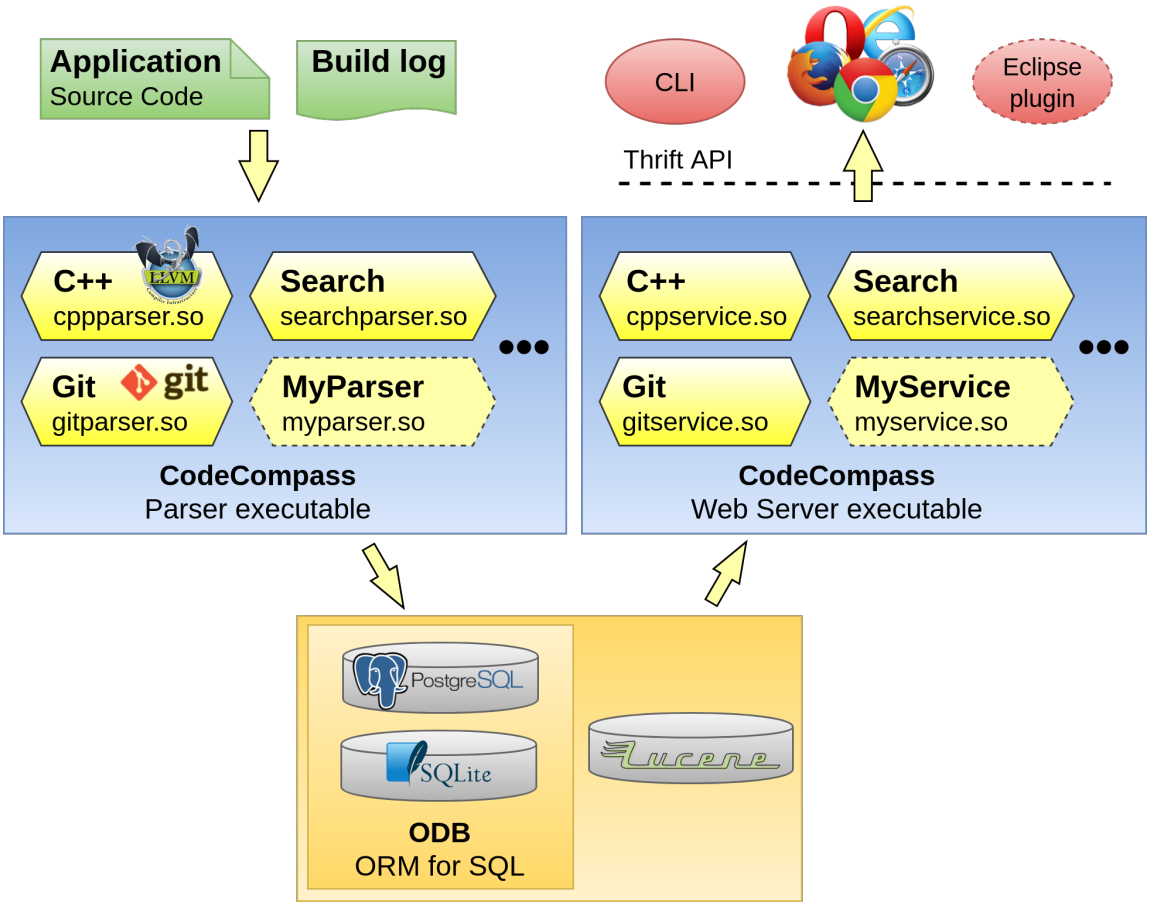


Figure 1: The CodeCompass architecture

which only interprets the source code textually, indexes it and tries to heuristically determine the type of each language element. This allows it to distinguish between the names of functions, variables or types, the definitions of names and references, but they have limitations, for example, they usually unable to execute overload resolution. The lack of semantic analysis allows it to support many programming languages. The information storage is highly optimized for searches, so OpenGrock has a fast response time in most cases. In return, the user often has to resolve ambiguities themselves, which is not a trivial task in large projects, and a possible mistake can lead to serious misunderstandings.

Woboc Code Browser [15] is also a web-based system, but unlike OpenGrock, it performs deep, semantic analysis. The user can quickly find files, classes, or other named program elements using a quick search that also includes code completion. Navigation is done using static HTML pages generated during the analysis, which is clearly advantageous in terms of operating speed. Hovering over individual elements displays additional information about the program elements. For example, in the case of functions, we can see the signature, definition, and call locations; in the case of classes, the size, data members, and possible inheritance relationships; in the case of variables, the type, and write or read points. Woboc handles the expansion of C/C++ macros well. Woboc achieves all this deep information by using a current compiler, the LLVM/Clang environment, for analysis, unlike OpenGrock. Of course, this also means that Woboc is limited to C and C++ projects.

Understand [16] is not only a code viewer and navigation tool, but also a complete IDE installed on a computer, meaning that the developer can modify the code and the changes are immediately reflected in the tool. In addition to the functionalities mentioned above, Understand provides a number of statistics and metrics about the system under study. In addition to the usual line-based metrics, such as total, average, and line-by-class, we also find basic structured and object-oriented metrics (such as cyclomatic complexity, coupling, and lack of cohesion)[17, 18]. Visualization often uses so-called Treemap, which

shows hierarchical relationships with embedded tiles and metric values with colors and tile sizes. In large projects, it is also important to examine the general architecture of the system. Understand can show various dependency relationships, function call chains and many other diagrams. Understand creates a database of the collected information during the analysis and all this data is accessible to the user through a programming interface.

CodeSurfer [19] is also a "thick" client like Understand and targets C and C++ (as well as x86-based machine code) projects. Similar to Woboc and Understand, it also performs deep semantic analysis, including some pointer analysis and impact analysis. The latter, for example, shows which instructions depend on a selected program point, which requires serious data flow analysis.

4. The CodeCompass architecture

CodeCompass [3, 20] is an open source, free software comprehension framework developed jointly by Eötvös Loránd University and Ericsson Hungary Ltd. The general architecture of the framework is shown in Figure 1.

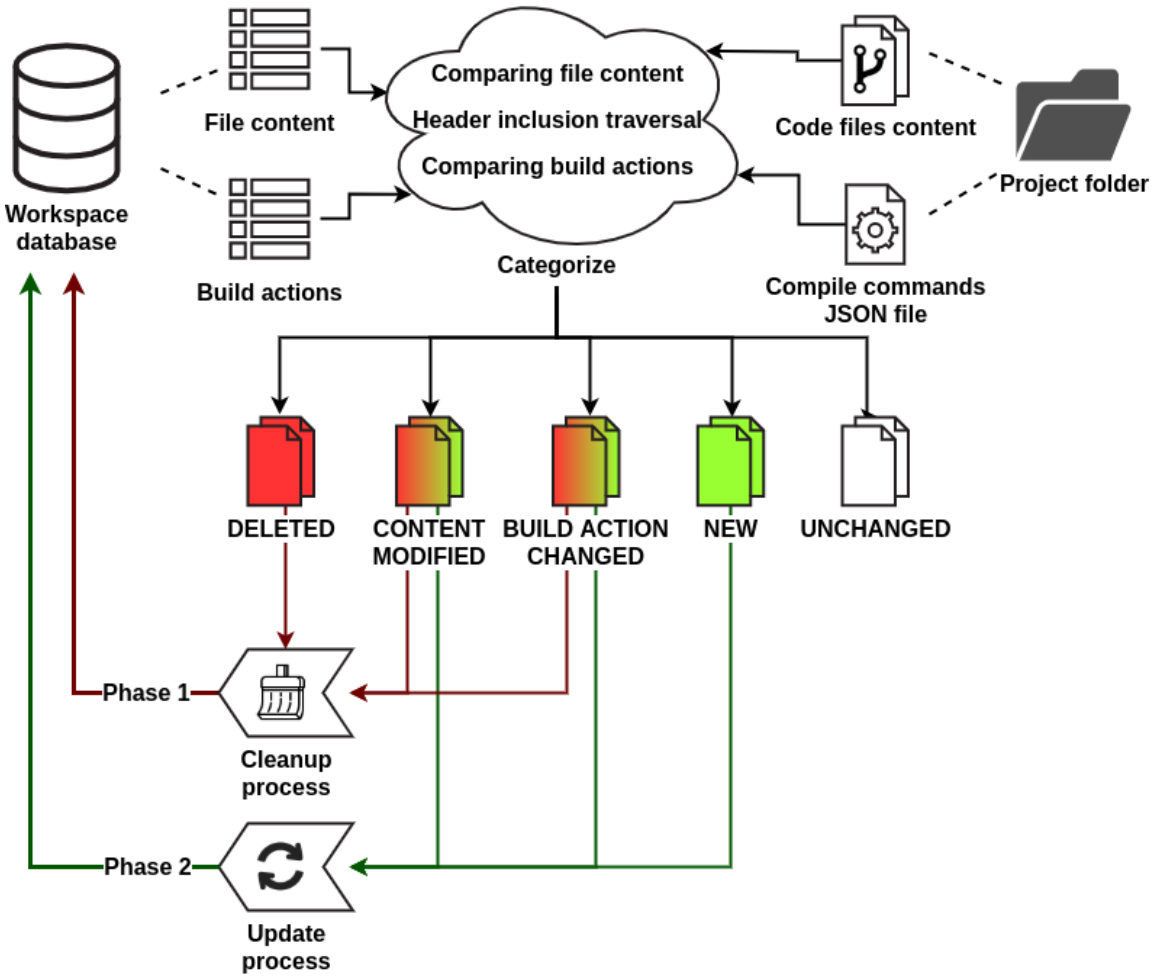


Figure 2: Incremental workflow in CodeCompass

CodeCompass has two main stages: collecting information about the project under review ("parsing") and publishing this information to the client(s) via a REST API. Accordingly, the two main binary components are the parser and the web server. The data collected by the parser is stored in a database, which is used by the web server. Both the parser and the web server can be extended with new functionality using plug-ins, which also contain the necessary database extension descriptions. The

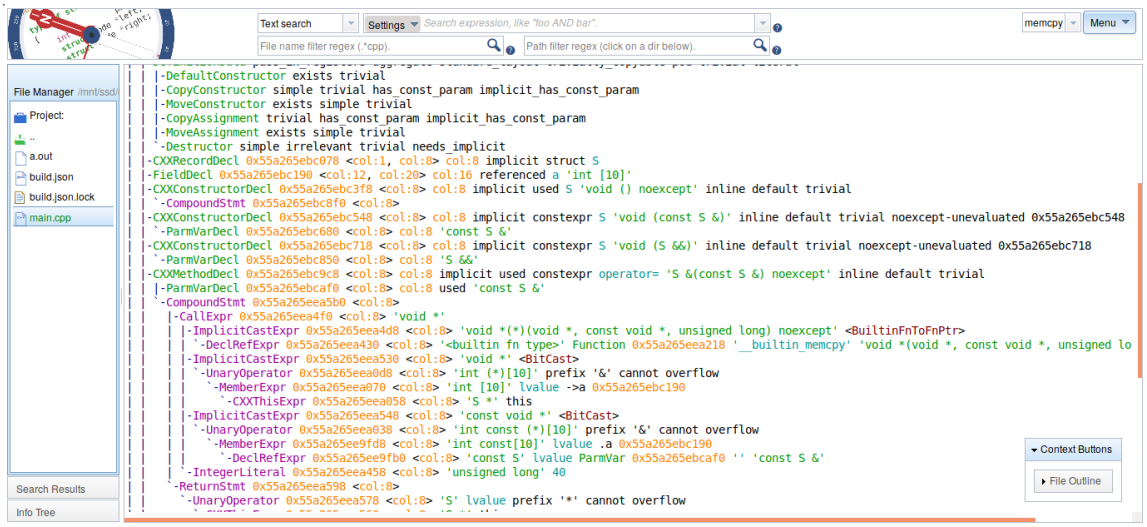


Figure 3: Typical screenshot with AST view

database is partly relational (e.g. postgreSQL), partly support fast search (e.g. Lucene), and, if available, CodeCompass also stores the version control (e.g. git) database related to the project.

The workflow is as follows. The parser (via plug-ins) analyzes the project, often together with the build commands. In the case of C and C++ projects, this is done, for example, by the compile command json file, which can be extracted from the build system in the case of cmake-based projects, or, if this is not available, a special tool, the logger, can generate it while executing the build commands. An abstract syntax tree (AST) is created from each file, which the parser examines and stores the most important data in the database.

For languages outside of the C/C++ family, parsers are implemented in language specific ways, usually utilizing some existing AST parser. The Java programming language modules, e.g., are parsed using the open-source Javaparser software [21], while C# projects are using the Roslyn parser [22] from the .NET Compiler Platform SDK [23].

CodeCompass captures a single state of a software project, but it is also possible to switch between multiple versions. If the code changes, it is possible to update the databases incrementally, as shown on Figure 2. The parser detects changes in the source files, even if this is done implicitly, by changing the contents of a header file. It then automatically determines the impact of the modification on the database and only updates the appropriate contents [24].

The web server provides information to the clients via a REST API. The client can be a user who accesses the web server through a standard browser, a device, such as an editor, that communicates with CodeCompass through an add-on, or a purpose-built software. The stateless interface allows the server to scale well and serve clients on multiple threads at the same time. CodeCompass aims to complete as many queries as possible within 1 second, as experience has shown that if the wait time is longer, the user tends to interrupt the workflow and starts another activity.

5. User interface and typical workflow

The CodeCompass user interface is divided into three main parts. As shown in Figure 3 , the search interface and options are at the top, the navigation panel is on the left, and the central panel occupies the largest part of the interface, in which the currently examined code snippets or diagrams are displayed.

A typical code comprehension session usually starts with the localization of the program element we are looking for. There are several ways to do this. If we know in which header or source file the element is located, we can use the usual navigation panel to load the file. If this information is not available, we can search for the name of the file, class, function or variable using regular expressions.

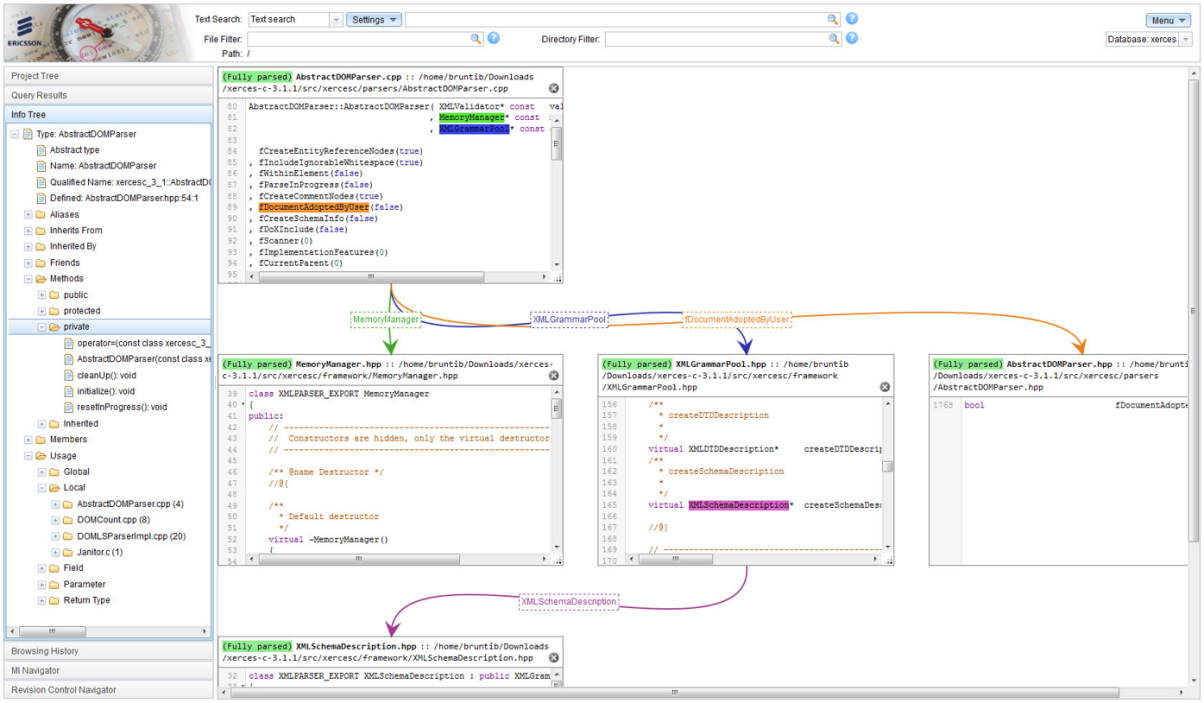


Figure 4: CodeBites view providing flexible fast code browsing

The search shows the results ranked by page-rank. We can include the full text of the source files and all files in the search, but we can also exclude, for example, comments, and can filter for different types of language elements.

One interesting search option is the “log search”. Programs often emit messages that cannot be found one-for-one in the source file, since they are generated from the values of certain components (e.g. file name, program line, error code). The log search determines the points (also displayed based on page-rank) that are the most likely sources of such messages. Log search is extremely effective when searching for the cause of an error displayed in a message or when investigating the software system based on a log file.

If the file you are looking for is found, its source code is displayed in the central panel. By clicking on a program element with the mouse, the most important information is displayed in the so-called information panel on the left, and by right-clicking, a menu depending on the type of element is displayed. In addition to the usual “jump to element definition” option and the information panel, we can choose version management information, short summary documentation, language-specific functions and the display of various diagrams.

Diagrams, which are powerful tools for visual representation, can be used organically within CodeCompass. The architecture diagram shows the relationships between system components, there exist the function call chain and class diagrams, and the CodeBites diagram enabling fast code browsing all be used interactively. CodeBites (see Figure 4) is a special visualization where elements referenced in the code (variables, functions, types, macros) can be displayed side by side, even if they otherwise exist in different files. CodeBites is the opposite of the bento-box technique; it allows fast, unstructured browsing involving a large number of program elements, which could all be in different files.

The further development of CodeCompass is ensured by plug-ins, which can include parser, server, web client and database elements. Further extensibility is provided by the Language Server Protocol (LSP) interface [25], through which CodeCompass can be connected to any LSP-capable editor or standalone tool.

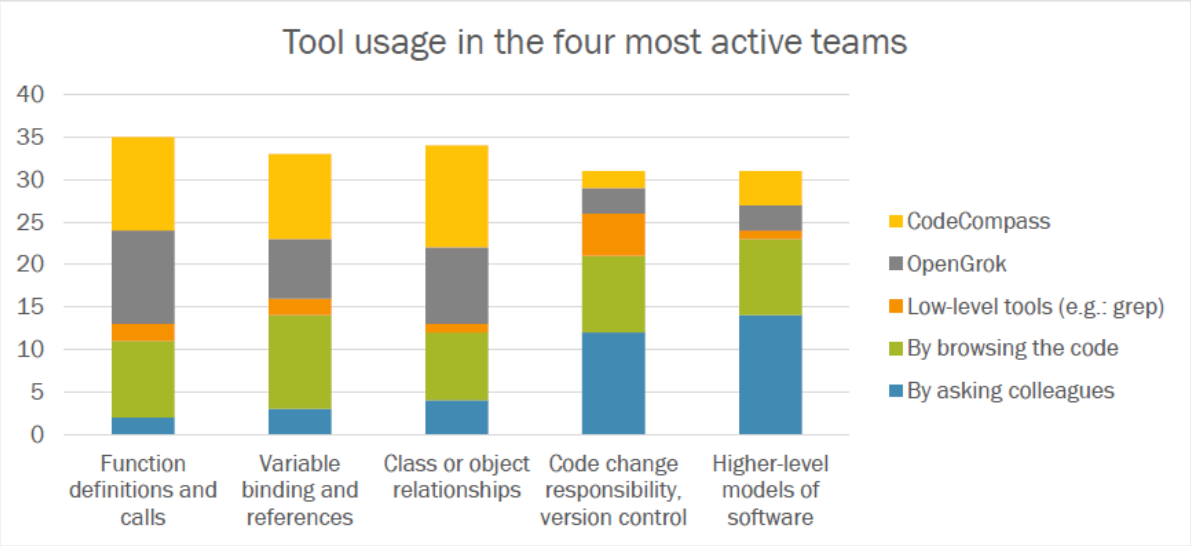


Figure 5: Comprehension tool usage in the most active teams

6. Industrial experiences

In this section we report the experiences of the industrial usage of CodeCompass in a multinational software company. The company has research and development sites located in Hungary, so we had first hand experiences about the introduction and everyday use of CodeCompass.

The company, working in the telecom industry, introduced CodeCompass in a gradual way. Developer teams integrated the tool into their CI/CD pipelines in a completely voluntary bases, many times together with other comprehension tools, like OpenGrok. Low level tools, like `grep` and similar were also used by developers. Within 5 years, the company had 500 CodeCompass users, from which about 10 percent used the tool multiple times a day. The volume of the analyzed code exceeded 20 million lines (mainly C and C++ code).

On Figure 5 we see how developers in the four most active teams (in number of commits) used comprehension tools in their daily work. We can see that CodeCompass was used mainly for determining class and object relationships and the usage of variables, but many times function definitions and function call chains were also inspected by CodeCompass. One interesting (and somehow surprising) aspect revealed that the most frequent tool used by architects to map high level software structure was CodeCompass.

7. Conclusion

The software industry is constantly working under pressure to be efficient, while the complexity of software is increasing, labor costs are rising, and deadlines are tightening. Meanwhile, developers spend a significant (sometimes the largest) part of their time supplementing or improving existing code, for which a perfect understanding of the software environment is an essential prerequisite. Traditional development environments are not optimized for understanding code. CodeCompass, as a web interface-based, plug-inable, open source framework, has successfully proven the right to exist for purpose-oriented software supporting code understanding. Over the past years, continuous improvements based on user feedback and the introduction of new functionalities (new programming languages, software metrics, etc.) have resulted in a mature system that is free to use and worth to apply by the software development community.

Declaration on Generative AI

During the preparation of this work, the author did not use any AI tool.

References

[1] M. Enderin, D. LeCorney, M. Lindberg, T. Lundqvist, Axe 810—the evolution continues, Ericsson Review 4 (2001) 10–23.

[2] E.-A. Karlsson, L. Taxen, Incremental development for axe 10, in: Software Engineering—ESEC/FSE'97, Springer, 1997, pp. 519–520.

[3] Codecompass, 2026. URL: https://github.com/Ericsson/CodeCompass.

[4] A. Von Mayrhauser, A. M. Vans, Program comprehension during software maintenance and evolution, Computer 28 (1995) 44–55.

[5] M.-A. Storey, Theories, methods and tools in program comprehension: Past, present and future, in: 13th International Workshop on Program Comprehension (IWPC'05), IEEE, 2005, pp. 181–191.

[6] M. P. O'Brien, Software comprehension–a review & research direction, Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report (2003).

[7] R. Brooks, Towards a theory of the cognitive processes in computer programming, International Journal of Man-Machine Studies 9 (1977) 737–751.

[8] E. Soloway, B. Adelson, K. Ehrlich, Knowledge and processes in the comprehension of computer programs, The nature of expertise (1988) 129–152.

[9] N. Pennington, Comprehension strategies in programming, in: Empirical studies of programmers: second workshop, Ablex Publishing Corp., 1987, pp. 100–113.

[10] O. Levy, D. G. Feitelson, Understanding large-scale software: a hierarchical view, in: Proceedings of the 27th International Conference on Program Comprehension, IEEE Press, 2019, pp. 283–293.

[11] A. Marcus, A. Sergeyev, V. Rajlich, J. I. Maletic, An information retrieval approach to concept location in source code, in: 11th working conference on reverse engineering, IEEE, 2004, pp. 214–223.

[12] V. Rajlich, N. Wilde, The role of concepts in program comprehension, in: Proceedings 10th International Workshop on Program Comprehension, IEEE, 2002, pp. 271–278.

[13] Opengrok, 2026. URL: https://opengrok.github.io/OpenGrok.

[14] Doxygen, 2026. URL: http://ctags.sourceforge.net.

[15] Woboq, 2026. URL: https://woboq.com/codebrowser.html.

[16] Ctags, 2026. URL: http://www.stack.nl/~dimitri/doxygen/.

[17] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (1976) 308–320.

[18] B. Henderson-Sellers, Object-oriented metrics: measures of complexity, Prentice-Hall, Inc., 1995.

[19] Codesurfer, 2026. URL: https://www.grammatech.com/products/codesurfer.

[20] T. Brunner, M. Cserép, A. Fekete, M. Mészáros, Z. Porkoláb, Towards better tool support for code comprehension, in: Z. Porkoláb, V. Zsók (Eds.), Composability, Comprehensibility and Correctness of Working Software, Springer International Publishing, Cham, 2023, pp. 165–201.

[21] Javaparser community, Javaparser project, 2026. URL: https://javaparser.org.

[22] Microsoft, Roslyn parser, 2026. URL: https://github.com/dotnet/roslyn.

[23] Microsoft, The .NET Compiler Platform SDK, 2026. URL: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/.

[24] A. Fekete, M. Cserép, Incremental Parsing of Large Legacy C/C++ Software, in: 21th International Multiconference on Information Society (IS), Collaboration, Software and Services in Information Society (CSS), volume G, 2018, pp. 51–54.

[25] Microsoft Corporation, Language Server Protocol Specification, Technical Report 3.18.0, 2026. URL: https://microsoft.github.io/language-server-protocol/specification.