MIPRO 2019, May 20-24, 2019, Opatija Croatia

# Delivering comprehension features into source code editors through LSP

Mónika Mészáros
*Dept. of Programming Languages And Compilers*
*Eötvös Loránd University*
Budapest, Hungary
bonnie@inf.elte.hu

Máté Cserép, Anett Fekete
*Dept. of Software Technology And Methodology*
*Eötvös Loránd University*
Budapest, Hungary
{mcserep,hutche}@inf.elte.hu

*Abstract*—**The maintenance of large, legacy software often results in higher development time and cost due to increasing size and complexity of the codebase and its documentation, their continuously eroding quality and fluctuation among developers. Code comprehension tools are designed to tackle this issue by providing various textual information, visualization views and source code metrics on multiple abstraction levels. These tools usually process not only the codebase, but also the build information, the version control repository and other available information sources. Meanwhile source code editors and integrated development environments (IDEs) are not performing well in the field of code comprehension as they are optimized for writing new code, not for effective browsing. This can easily result in frequent switching between environments during development, hindering effective programming and raising development cost. Language Server Protocol (LSP) is an open-source protocol to connect source code editors with servers that provide language-specific features. In this research we analyze how LSP can be utilized to improve the code comprehension experience inside code editors by integrating the features of such tools through remote procedure calls. As a prototype solution we showcase the integration of two open-source applications: Visual Studio Code and the CodeCompass code comprehension tool.**

*Index Terms*—**code comprehension, software maintenance, language server protocol, source code editor**

## I. Introduction

The lifetime of large legacy software systems can often span over a decade, resulting in several maintenance difficulties. Due to the generally high fluctuation among developers in the software industry, not only the size of the software increases, but a large number (i.e. hundreds) of programmers get involved in the project. During the extended development time, original specifications and design intentions can get lost, the documentation can get unreliable and the code quality can deteriorate. Thus methods and tools to support newcomer programmers to comprehend the existing codebase (or at least the relevant modules) and to reduce the required timespan until they can effectively join the workforce became a substantial aspect of the software development industry.

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

Source code editors in general are intended to help the creation of new code with such features like syntax highlighting, code completion, code navigation, code generation, syntactical and semantic error checking and refactoring. Integrated development environments (IDEs) further improves the development experience by integrating the compiler, debugger, tester, version control and other tools used in the development cycle to form a single environment capable to deal with multiple or even all tasks of software development.

While some code editors and IDEs are equipped with advanced code comprehension features [1], most of them are not prepared to help developers navigate in larger codebases. Experience shows that joining into the development of a long-running project is much more common than starting to write a new software from scratch. This means that developers would usually need extended comprehension features of their preferred editor while trying to understand the codebase they are about to modify or broaden.

Extending IDEs with their lacking code comprehension features (e.g. through plugins) means that code of the same purpose had to be written for different platforms which results in redundancy and maintenance difficulties. Code comprehension tools and code editors also lack a common protocol for communication with each other. While some research on IDE independent approaches have also been evaluated by injecting comprehension information into the source code itself as comments [2], missing proper visualization degrades the developer experience. Our research therefore focuses on the integration of existing code comprehension tools into IDEs through the LSP standard.

## II. Code comprehension and code editors

This section covers the comparison of a couple of well known editors to determine what features are generally missing or present in them regarding code comprehension.

### A. Abilities and deficiencies of editors

Source code editors often include a set of simpler features designed for code comprehension. Tab. I shows a set of

TABLE I
THE ABILITIES AND DEFICIENCIES OF WIDELY USED IDES

| IDE | Go to definition | Find all references | Dependency graphs | UML diagrams | Symbol search | Code completion |
|---|---|---|---|---|---|---|
| Visual Studio Code | ✓ | ✓ | ✓ | $P$ | ✓ | ✓ |
| JetBrains IDEs | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Eclipse | ✓ | ✓ | $P$ | ✓ | ✓ | ✓ |
| Qt Creator | ✓ | ✓ | $P$ | ✓ | ✓ | ✓ |
| Atom | ✓ | ✓ | $X$ | $P$ | $X$ | ✓ |
| Emacs | ✓ | ✓ | $X$ | $P$ | ✓ | ✓ |
| Sublime Text | ✓ | ✓ | $P$ | $P$ | ✓ | ✓ |
| vim | ✓ | ✓ | $X$ | $P$ | ✓ | ✓ |

IDEs and their capabilities and deficiencies regarding code comprehension tools[1].

**Go to definition** It's one of the most common code comprehension supporting features. Developers usually need to take a look at a function's, attribute's etc. original definition, e.g. to obtain knowledge of field or formal parameter types or method signatures. The feature can be found in almost all modern editors.

**Find all references** Its frequency is similar to that of *Go to definition* as it is also a very basic tool of code comprehension. This feature is extremely useful when one tries to understand a component of a software since developers usually need to know where each and every field is used in the code.

**Dependency graphs** A very useful tool when a programmer tries to understand the structure of a legacy project. The larger the software, the more helpful this feature becomes as it depicts the connection between different files and entities of the software in a simple, transparent form. It's a less common feature in IDEs.

**UML diagrams** The diagrams that describe classes, packages, use cases or even individual objects and the connections between them are used by developers to understand complex software architectures. They are also used in reverse engineering when developers try to decrypt the structure of a software without knowing the code itself. It is worth noticing that IDEs might support the generation and the manual creation of UML diagrams as well.

**Search** Useful for both writing and understanding code. The broader the search options, the more help it provides. Searching might cover one or more chosen files, may be filtered to e.g. only comments or actual code and search parameters might be set to target the full given text or subtexts. Symbol searching is useful in exploring code with semantic meaning.

**Intelligent code completion** Although first and foremost useful in writing code, this feature is also helpful in understanding the smallest bits of code. By hovering above an identifier, its usages and possible corresponding methods and attributes are shown, thus helping the developer work out the structure of the codebase.

[1] ✓ symbolizes has a built-in version of the feature in the column, $X$ means that the given feature is missing from the IDE, and $P$ symbolizes that the IDE has the feature in form of a plugin.

## B. *Code comprehension tools*

While common code comprehension features described in Section II-A provide the expected code navigation and understanding support during development, the comprehension of large legacy software for a newcomer developer without detailed preliminary knowledge of the system requires the aid of enhanced code comprehension tools.

**Advanced search** Code comprehension tasks often start with a fast feature location with the aim to narrow down the code segments where the features to be worked on are implemented. Beside the traditional text search, multiple features can improve this functionality, like: extending the search phrase with regular expressions and logical operators, symbolic search of a specific programming language, file name or path filtering based on a search pattern or fuzzy search – e.g. for log or error message search, where an exact match cannot be expected.

**Enhanced code browsing** During code comprehension, ad-hoc browsing the code should be made as straightforward and fast as possible. This can be supported by a browsing history view which shows the direction of navigation in a tree structure instead of a linear one, so the user may easily jump back and forth between the traversed locations of dependencies, e.g. function calls. Some tools also provide interactive views with multiple code displaying windows open with their traversed connections (function calls, variable definitions, etc.) visualized as connections – see *CodeBites* in CodeCompass.

**System architecture views** 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 [3]. By analyzing all the build information at compile time beside the source code we can differentiate between the types of relationships between the source, header and binary files. An implementation file *uses* a header file if it uses a symbol declared in the header, but it can also *provide* it, when defines the names declared in the header. There can be *containment* relation between binaries and source files: a binary may contain several source files if those are linked together in the build phase. Similar views are of great interest also on a higher, module (directory) level. Through visualizing these relations of system architecture (e.g. like on Fig. 1), the complex connections of a software are more easily perceived [4].

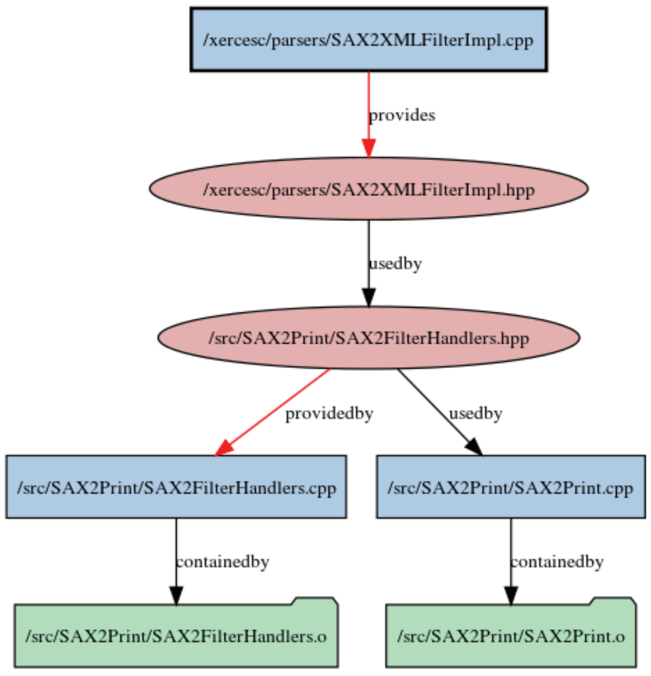
/xercesc/parsers/SAX2XMLFilterImpl.cpp
provides
/xercesc/parsers/SAX2XMLFilterImpl.hpp
usedby
/src/SAX2Print/SAX2FilterHandlers.hpp
providedby
usedby
/src/SAX2Print/SAX2FilterHandlers.cpp
/src/SAX2Print/SAX2Print.cpp
containedby
containedby
/src/SAX2Print/SAX2FilterHandlers.o
/src/SAX2Print/SAX2Print.o

Fig. 1. User components of `SAX2XMLFilterImpl.cpp` in *Xerces-C++*.

**Revision control navigation** A main resources of knowledge about the a projects' source code is hidden in version control commit messages. Therefore alongside the commit messages for a file, presenting the author for each line of code and visualizing how old a code segment is (e.g. through color coding) can also aid code comprehension.

## III. Language Server Protocol

Language Server Protocol is a joint project of Microsoft and others to standardizes the communication between language tooling and code editor. In other words, LSP comes as a new layer between IDEs and language tooling. This enables LSP compliant tooling services and IDEs to communicate in a standard way, thus making them independent of each other and providing the following advantages: language tooling services can be implemented in any language making it possible to adapt to special needs and/or optimize the performance of these utilities. At the same time, since code editors are also independent of the tooling environment, language tooling can be assigned to multiple IDEs and also an IDE can make use of several LSP compliant language tooling utility [5].

The Language Server Protocol is a very useful approach that supports the development of tools that help to make source codes more understandable. However it lacks certain functions that could enable it to get to the next level.

LSP is based on client-server mechanism, where the client can send requests to the server, and it can display the responses of the server in the development environment (IDE). Responses can be classified into 3 groups:

- returning position or position list: e.g. response for a *Goto Definition* request is the position of the definition, or response for a *Find all References* request results in a list of reference positions
- returning information as text: information inserted in the document or information displayed as a tooltip
- executing (a) command(s): for certain predefined request types a command that calculates or displays something – can be set to be executed

Typically LSP returns responses containing texts (position, short texts, or perhaps a name of a command), which can be freely used by the integrating code editor to display the information, either in a textual or visual way [6]. Still there is demand for more complex structures/types as response content. Imagine a scenario where clicking on the name of a class the server would return the UML diagram of the class in picture format. An even more advanced feature could be a response containing an interactive UML diagram, where clicking on certain parts of the picture would trigger events to which event handler methods could be assigned.

While the *executive command support* feature is useful, providing an option for the user to be able to pick an executable command themselves would further improve usability. Implementing a response type that returned a list of commands depending on the position in the file that could be executed on that position (e.g. diagnostic tasks or displaying specific diagrams), would further enhance the customizability of the protocol. These could be displayed in a context menu of the IDE and upon choosing one would execute the corresponding command.

Currently it is not supported (or at least it is not easily implemented) that the server and the client run on a different machine. This could prove useful in such cases where processing of requests is quite demanding as it requires significant amounts of resources, therefore it is not fortunate to let the client deal with it.

## IV. Integration into CodeCompass

To demonstrate the capabilities of LSP, we implemented its methods into the backend of CodeCompass [7]. CodeCompass is an open source code comprehension supporting software developed by Ericsson Ltd. in cooperation with Eötvös Loránd

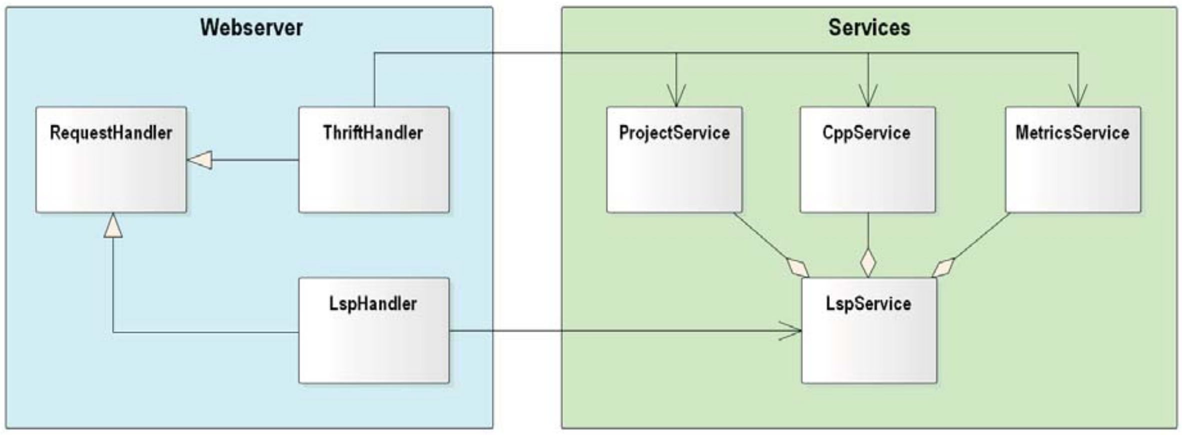

Fig. 2. Class diagram of the LSP integration into CodeCompass.

University, Budapest. It provides various navigational functions and textual search about the processed code by not only relying on the codebase but by obtaining information from the build system. The latter is done by processing the compilation database of the software [8] and using the JSON file that contains the compilation commands.

CodeCompass possesses a web user interface which provides interactivity by depicting clickable dependency diagrams that navigate the user through the modules, classes and methods of the software. Our purpose was to expand the capabilities of CodeCompass and support its usage in external source code editors. Visual Studio Code became the showcase of this experiment since it is also an open source software that supports the creation of plugins. LSP was used to connect CodeCompass to VSCode.

Functionalities in CodeCompass are organized into services (*ProjectService*, *CppService*, *MetricsService*, *SearchService*, etc.), being responsible for the features their names suggest. Their public API is served by the webserver component through Thrift protocol [9]. Since LSP requests are cross-compliant with lower level functions in CodeCompass, we defined a wrapper service which combines the existing functionalities for LSP.[2] This wrapper is named `LspService`.

Thrift and LSP use different constructions for their requests, therefore a new request handler module named `LspHandler` was also introduced to manage the transformation between LSP messages in JSON format and the strongly-typed inner representation of the program, as shown in Fig. 2.

## V. Integration into Visual Studio Code

According to the Extension Guide [10] the language server in VS Code consists of two components.

[2]For example requesting the definition location for a symbol is a single call in LSP, while with the lower level API of CodeCompass, first the AST node identifier has to be queried, resulting in two calls.

**Language Client** : a normal VS Code extension written in TypeScript (use `vscode-languageclient` library)

**Language Server** : a language analysis tool running in a separate process (use `vscode-languageserver` library)

### A. Context menu – context dependent dynamic menu items

Unfortunately LSP doesn't have a feature to place a context menu at an arbitrary spot in the document, where the user could choose from context dependent list of executable commands. Although the protocol supports the execution of commands for certain predefined request types (one command for each type), these commands can only be assigned to previously defined spots in the document (e.g. see Code Lens). All IDEs supporting LSP have context menus, these cannot be extended freely. New Menu items are automatically created and displayed when a supported request type is implemented.[3] However it is not possible to customize through the protocol.

Despite the lack of support for full customization in the protocol, the feature can be implemented without changing the VS Code integration modules – client and server – or enhancing the LSP protocol, by a non standard usage of the code completion feature. By taking advantage of this possibility a dynamic context dependent menu can be created (menu item types and quantity depends on the the environment where it was effectuated)

Basics of implementation:

- Code completion can also be initiated by a trigger character (in case of such use of code completion the `context.triggerKind` and `context.triggerCharacter` properties of the `CompletionParams` parameter are filled). Listing the context dependent menu items / commands can be

[3]E.g. in case of VS Code *goto definition* or *find all references* items can be displayed in the context menu when they are implemented.

achieved by setting the trigger character to a rarely used character, and by basically overriding the standard behavior based on a condition of the trigger character field's content. If the code completion had been initiated by the special character, instead of displaying the normal list of keywords for code completion, our own special method takes over, and a list of executable commands can be returned to the context menu. The list of available commands can be based on the position in the code as well (position property of the request parameter).

- Documentation can also be provided for menu items by supplementing the `detail` property of the `CompletionItem` property of the response for better understanding.
- Opening such context menu would normally insert a code piece into the source code (since this is basically still a code completion feature), however this insertion can luckily be avoided by customizing code completion in such manner that no actual insertion would take place, what more the special character triggering the special context menu can also be deleted (see: `textEdit` and `additionalTextEdits` properties of `CompletionItem` property).
- Furthermore LSP provides functionality to set a command that would be executed after selecting a menu item (see: `command` property of `CompletionItem` property).

Overall using this method for creating a context sensitive dynamic menu enables the IDE to display a position based list of custom menu items at arbitrary places in the source code without actually entering any new codes and at the same time executing a command based on the selected menu item. On top of this the custom created menu triggered by the special character and the code completion functionality stays independent of each other, so in the end the features have been extended without losing any standard functionality.

In our solution we use the context menu to be able to select from a list of diagrams provided by CodeCompass. Triggering the menu on a specific place in the code would send a request from LSP to CodeCompass to get the list of appropriate diagrams specific to that code context.

## B. Displaying diagrams

LSP does not support displaying pictures (diagrams) on selecting a code piece, therefore we created new request type which has the following request parameter:

- in case of diagrams related to the whole document

```
interface WholeFileDiagramParams {
  textDocument: TextDocumentIdentifier;
  diagramType: string;
}
```

- in case of diagrams specific to position in the code:

```
interface PositionDependentDiagramParams {
  textDocument: TextDocumentIdentifier;
  position: Position;
  diagramType: string;
}
```

Response is the diagram in SVG format.

If the user selects a certain diagram type from the context menu list, the Language Client will send the server this specific diagram type, the file ID, and - in case of position specific diagram - the current position in the file. The server returns the diagram in SVG format, that will be saved to the file-system of the client as a picture - as VS Code does not handle SVG a conversion occurs here to PNG -, and finally the client will display the diagram (the converted PNG picture) in VS Code.

## C. Remote Language Server

LSP's model does not support remote servers. There are no such parameters among the initialization settings that would specify the location of the server (e.g. IP address, port). The definition in the protocol only specifies that the language server must be a separate process. Since it is not supported in LSP an attempt to create a remote server without enhancing the protocol itself would most probably result in custom solutions that are specific to a given code editor and a language server (due to the lack of standardization in the middle layer - LSP). Although some experts say that the remote server feature of language tooling is not where the science is heading currently, there are still applications where it is more or less inevitable. A transparent way of implementing such feature within LSP's client settings could be located e.g. in the `InitializeParams` interface:

```
  serverip: string
  serverport: int
```

The reason why the client must contain these parameters is simple: since the language server is started by the client, and the language client send requests to the server, it has to know where the server resides. Also as can be found in the documentation, server lifetime is managed by the client: *"It is up to the client to decide when to start (process-wise) and when to shutdown a server"* [5].

In our solution we used the "official" language server as a proxy towards the CodeCompass. As a bonus on top of the previously mentioned advantages, required conversions could be performed during the process (e.g. CodeCompass starts counting rows and columns from 1 whereas VS Code starts from 0).

As a demonstration for the interaction between the language server and client of VS Code and the LSP components in CodeCompass introduced in Sec. IV, the sequence diagram depicted in Fig. 3 shows how a component diagram produced by CodeCompass displayed inside Visual Studio Code upon the request of the user:

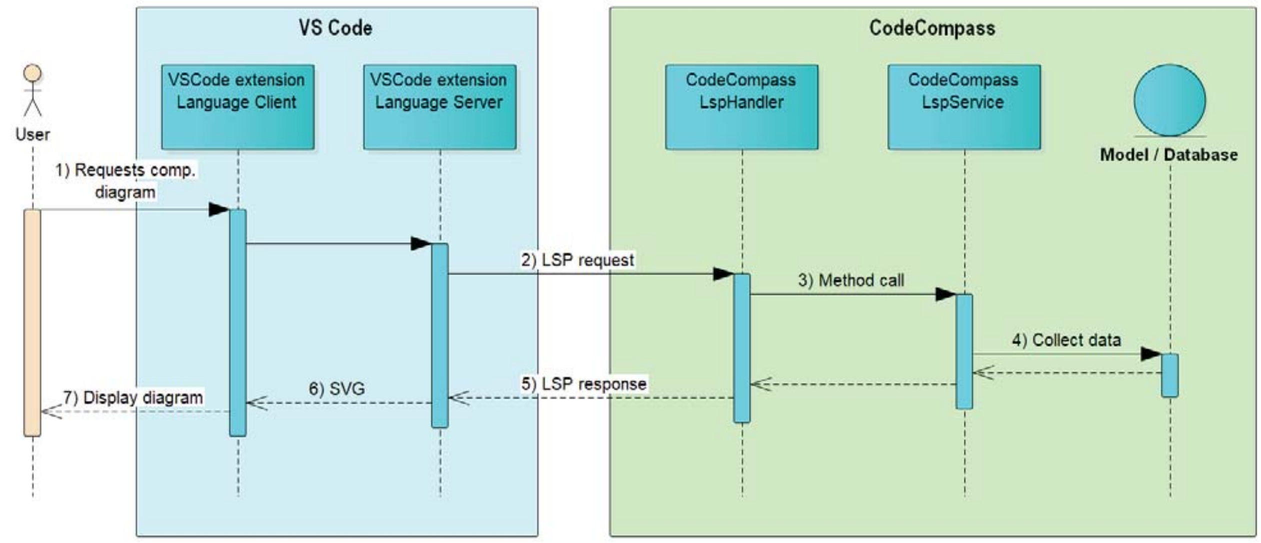

Fig. 3. Interactivity between VS Code and CodeCompass components to serve a diagram request.

1) The request for the diagram is issued by the actor through e.g. a context menu.
2) The language server constructs the corresponding LSP request towards the CodeCompass webserver.
3) The `LspHandler` component in CodeCompass processes the request and performs the necessary calls on the `LspService` component.
4) The required data is pulled from the underlying model layer (and ultimately from the workspace database) to build the diagram.
5) The result diagram is returned wrapped in an LSP response message.
6) The SVG diagram is extracted from the response.
7) The requested diagram is displayed to the user.

## VI. Conclusions and Future Work

Maintaining and developing legacy software is a general challenge software industry faces nowadays and most likely in the future. By delivering features of state-code comprehension tools into popular code editors, companies can ease the work for newcomer programmers, meanwhile reducing the development time and cost. In our paper we demonstrated a prototype integration of the CodeCompass backend and the Visual Studio Code IDE through LSP in a robust, reusable and cross-platform way.

Future work will extend the range of supported IDEs to further LSP compliant editors, e.g. Eclipse and vim. To provide an improved comprehension feature set, we also aim to integrate the incremental parsing [11] feature of CodeCompass into mentioned editors to provide real-time reanalysis of the modified source code.

## References

[1] D. Asenov, P. Müller, and L. Vogel, “The ide as a scriptable information system,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 444–449. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970329

[2] M. Sulr, J. Porubn, and O. Zorick, “Ide-independent program comprehension tools via source file overwriting,” in *2017 IEEE 14th International Scientific Conference on Informatics*, Nov 2017, pp. 372–376.

[3] I. Biederman, “Recognition-by-components: a theory of human image understanding.” *Psychological review*, vol. 94, no. 2, p. 115, 1987.

[4] M. Cserép and D. Krupp, “Visualization Techniques of Components for Large Legacy C/C++ software,” *Studia Universitatis Babes-Bolyai, Informatica*, vol. 59, pp. 59–74, 2014.

[5] Language Server Protocol Specification. [Online]. Available: https://microsoft.github.io/language-server-protocol/specification

[6] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän, “Visual augmentation of source code editors: A systematic mapping study,” *Journal of Visual Languages & Computing*, 2018.

[7] Z. Porkoláb, T. Brunner, D. Krupp, and M. Csordás, “Codecompass: An open software comprehension framework for industrial usage,” in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC ’18. New York, NY, USA: ACM, 2018, pp. 361–369. [Online]. Available: http://doi.acm.org/10.1145/3196321.3197546

[8] R. Szalay, Z. Porkoláb, and D. Krupp, “Towards better symbol resolution for C/C++ programs: A cluster-based solution,” in *IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2017, pp. 101–110.

[9] Apache Thrift. [Online]. Available: https://thrift.apache.org/

[10] Language Server Extension Guide. [Online]. Available: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

[11] A. Fekete and 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)*, vol. G, 2018, pp. 51–54.