Call Graph Visualization for Visual Studio Code

Anahita Chavan
CS, Lund University, Sweden
an8168ch-s@student.lu.se

Jacob Johansson
CS, Lund University, Sweden
ja2732jo-s@student.lu.se

Abstract

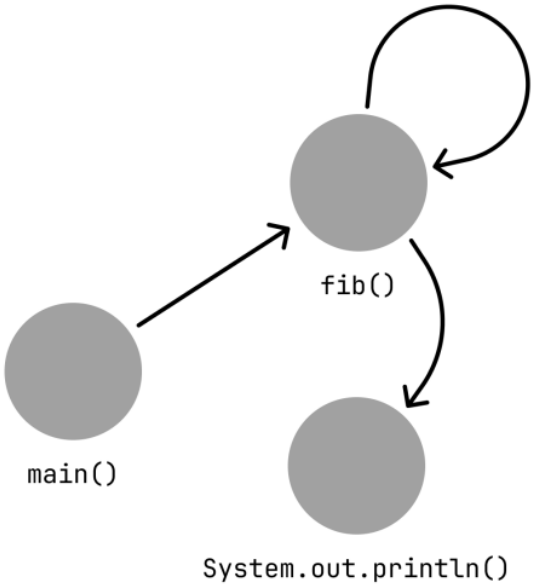
Understanding how parts within a codebase interact is vital for software development, especially in large and complex systems. This paper introduces an approach to call graph visualization as an integrated extension for Visual Studio Code. Our solution uses a client-server architecture, and leverages the Language Server Protocol, to provide demand-driven call graph generation for Java code. The tool has features such as selective file analysis, expandable nodes, and different depth levels of visualization. We evaluate our approach on simple and complex codebases; the solution has limitations in scalability and language support, which we address with proposed future work.

1 Introduction

It is crucial for a good understanding of a codebase to understand the relations between different parts of the code, how functions depend on each other, and how they interact. In complex systems, identifying performance bottlenecks, understanding how changes may affect the rest of the codebase, or debugging, can be quite difficult when the code is large with many moving parts. A code analysis tool can provide the developer with a greater understanding of a codebase, and how code spread across different files interacts.

One way to analyze code is with call graph analysis. A call graph [12] is a directed graph that shows relationships between different functions and other structures inside codebases. As shown in Figure 1, functions can be represented as nodes, with directed edges drawn between caller and callee functions. Recursive functions can be represented as a node, with an edge pointing at itself.

Being able to see a call graph while programming can be very beneficial. Call graphs can make the codebase more intuitive for the developer, as they provide a visual representation of how execution flows throughout different sections of a program. They can also be useful when working with new codebases that the developer may not be familiar with. They can additionally also make it easier to find dead code, specifically functions that are never called. In a call graph, these would be represented as free-floating nodes, with no connections to the main call graph. As a debugging tool, call graphs could be implemented to highlight relevant nodes as the developer steps through function calls.




```java
class Fibonacci {
  public static void main(String[] args) {
    fib(0, 1);
  }
  static void fib(int a, int b) {
    System.out.println(a);
    fib(b, a+b);
  }
}
```

Figure 1. Call graph with corresponding code

There is a gap between the tools that developers want in program analysis and the tools that exist, specifically tools that are integrated into development environments [2, 11]. Ideally, developers want a tool that is integrated into their code editor for maximum efficiency. Despite the potential usefulness of call graph analysis, there seems to be a rather limited amount of solutions. The current options for call graph analysis tools usually come packaged as stand-alone programs, and rarely stay synchronized with code updates. There is also a somewhat minimal amount of extensions for development environments that provide the sort of call graph analysis that we are looking for.

Course paper, EDAN70, Lund University, Sweden
January 16, 2025.

To integrate call graph analysis into a code editor, one could leverage language servers. A language server is an application which provides programming language specific information about source code, enabling features such as error highlighting, code suggestions, auto-completion and syntax highlighting. Language servers typically makes use of the Language Server Protocol (LSP) [8], a protocol developed by Microsoft that standardizes communication between code editors and language servers.

Before language servers and LSP, each editor had to implement language-specific features individually. This meant that if there was an m number of programming languages and an n number of editors/IDEs, each editor would need an integration for each language, resulting in $m * n$ integrations. LSP solves this problem by letting language servers implement language-specific features once, and by letting editors implement support for the protocol once. This results in $m + n$ implementations. Number of language servers (one for each language) + number of editors (one per editor) implementations (see Figure 2).

Figure 2. How LSP solves the $m * n$ problem

LSP and language servers save developer efforts for both editors and language tools by letting them focus on one integration with the protocol. This makes it easier to integrate existing tools with new editors or languages. LSP also ensures consistency by providing the same language features across different code editors.

This paper presents the following contributions:

- We present a solution to the problem of lack of integrated call graph analysis tools by developing an extension for a widely-used code editor (Our Approach).
- We evaluate our approach by testing it on different codebases, analyzing its feature set, limitations, and possible areas for future improvement (Evaluation).

Finally, we discuss related work (Related work) and conclude the paper (Conclusion).

2 Our Approach

To address the limited number of integrated call graph analysis tools, we developed a call graph visualization extension for Visual Studio Code, a widely-used code editor developed by Microsoft. Our tool uses a typical client-server approach, where the client is an application that requests services, and the server provides those services to the client. The client initiates communication with the server by sending it a request. The server listens for incoming requests, processes them, and sends responses back to the client. In our approach, the Visual Studio Code extension is the client and the language server is the server. We named the client `cat-viz` and the server `cat-server`. For a visualization of how they interact, see Figure 3.

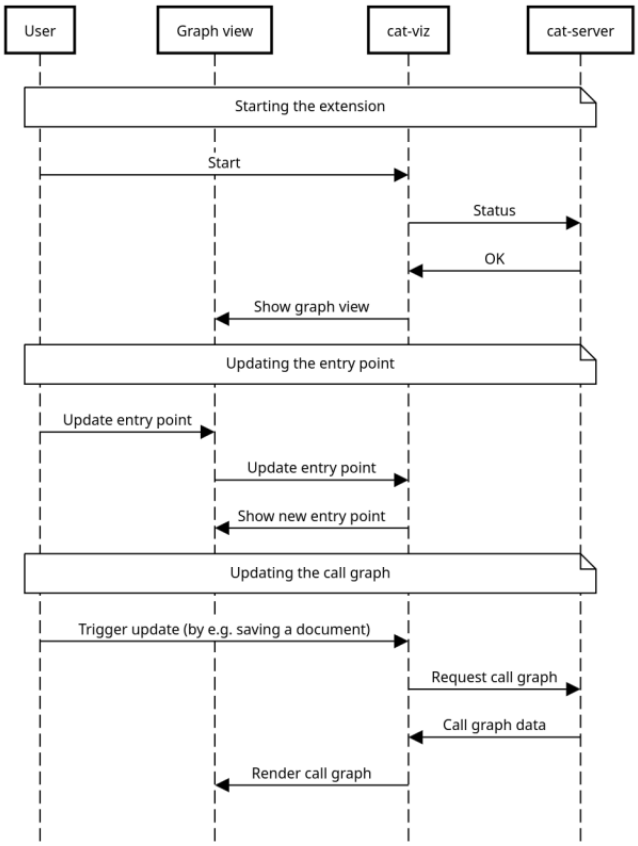


Figure 3. Sequence diagram showing communication between `cat-viz` and `cat-server`.

2.1 Demand-driven graph generation

We took precautions to ensure that neither the developer nor the server would get overwhelmed by a large amount of data. Generating a call graph representing a whole complex codebase could lead to performance issues, as a large amount of files would need to be analyzed. A large call graph could also make it harder for the developer to interpret the graph, as it may include a lot of unnecessary information. With these issues in mind, we chose to implement our solution using a demand-driven approach, where only the source code files that are specified by the user will be analyzed. This

enables users to selectively analyze the code that is relevant at the given moment, while ignoring everything else.

2.2 Server

We built the server by extending another project called CAT, short for Callgraph Analysis Tool [10], which is a call graph analysis tool for the Java programming language. CAT is built as an extension to the ExtendJ Java compiler [9] and functions as a stand-alone command-line tool.

ExtendJ generates an abstract syntax tree (AST), a tree-like structure that represents the structure of a program. ExtendJ is built using Reference Attribute Grammars (RAGs), a system for assigning attributes to nodes in tree-like structures, such as abstract syntax trees [5, 6]. These attributes are defined by equations, which are computed for each node. In the context of a compiler, RAGs can be used to create references between declarations and their corresponding uses. ExtendJ uses RAGs to link function calls to their respective function definitions, with each node representing a function call containing a pointer to the node representing the function definition. This in turn makes call graph analysis rather simple to implement, as it is just a matter of recursively identifying all function calls within a function body, retrieving the corresponding function body for each call, and repeating the process while storing each function call in a tree-like structure.

In order to integrate CAT into our solution, we decided that it would be expanded and turned into a language server. Instead of functioning as a stand-alone command-line tool, the server sits in the background and accepts incoming requests for call graph analysis from the client. Each request contains a list of source code files that the user wants to analyze. Upon a valid request the server will compute the call graph, and then pass the resulting data back to the client.

2.3 Client

The client is built as a Visual Studio Code extension. For displaying the call graph, we chose to use Cytoscape.js [4], which is an open-source graph visualization library written in JavaScript.

The extension has the following features, listed and described below:

- A sidebar component with the files that the user wants to analyze. The files that the user wants to analyze are called dependencies. `.java` files can be added as dependencies, and `.jar` files and directories can be added to the classpath. Dependencies are added through the file explorer through an option in the context menu. See Figure 4.
- Nodes can be either circle-shaped or diamond-shaped, depending on whether the method belongs to a file that is listed as a dependency. If the file is a dependency, the node is displayed as a circle. Otherwise, it is displayed as a diamond. See Number 5 in Figure 5.
- If a diamond-shaped node exists on our classpath, the user can right-click on the node and select "Expand". This will add the file with the corresponding method to the list of dependencies, and update the graph accordingly. Some diamond-shaped nodes cannot be expanded as they are derived from standard libraries or have unknown file locations. See Number 4 in Figure 5.
- Before a call graph can be generated, an entry point must be entered. The entry point is the method at the center of the call graph, which the call graph builds upon. The entry point can be changed at any point, see Number 3 in Figure 5.
- The user can choose to view the graph on a method level, class level, or package level. This feature allows the user to potentially gain a greater understanding of the codebase by showing not only the relationships between methods, but also between classes and packages. See Figure 6 and Number 2 in Figure 5. By right-clicking on a class or package, the "Expand" function can be used to expand to a lower level. See Number 4 in Figure 5.
- The user can highlight a node by left-clicking on it. Once a node is highlighted, the extension displays its forward call graph, showing all the nodes that are reachable from the highlighted node. The reachable nodes are highlighted in red, while the remaining nodes are grayed out. See Number 6 in Figure 5.
- The tool has a search feature, allowing the user to search for specific nodes. See Number 1 in Figure 5.

Figure 4. Sidebar component with list of dependencies

The communication between the code analyzer and code editor is done through a custom protocol that is built on top of LSP. The specifications for LSP do not include anything related to call graph analysis. We therefore had to fill

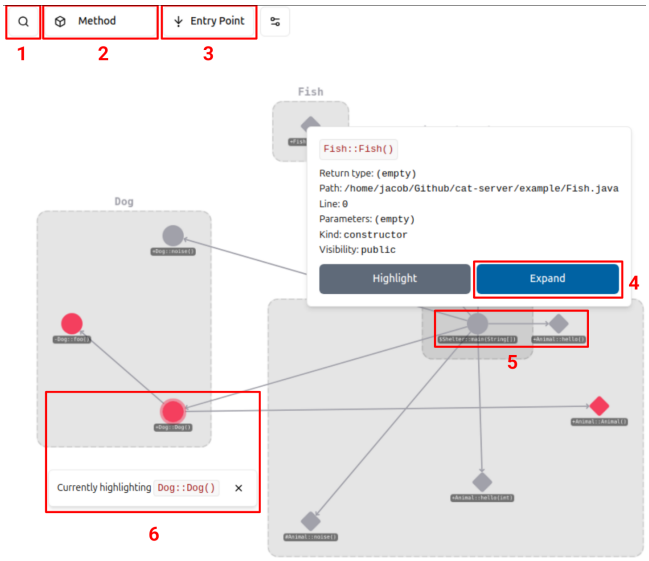


Figure 5. The call graph view

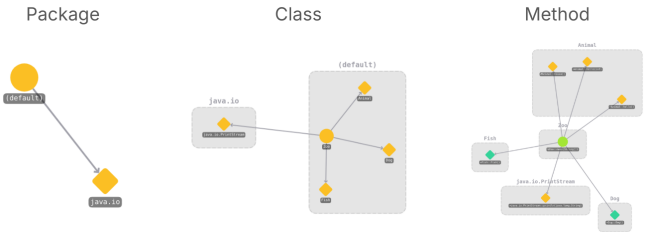


Figure 6. Different depth levels

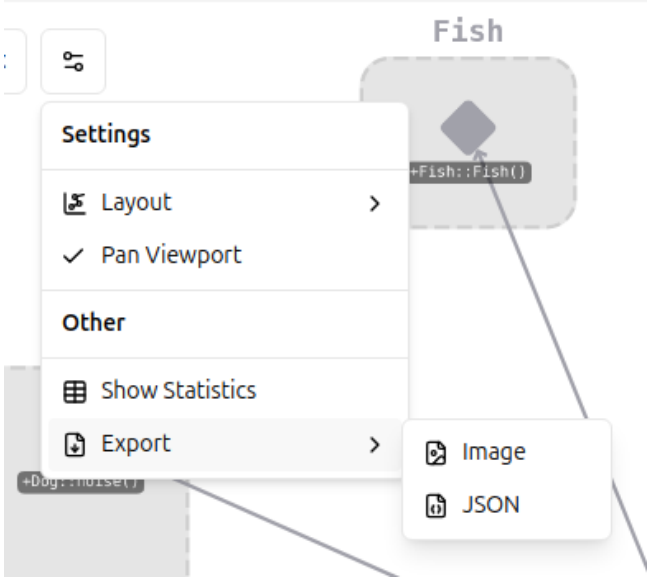


Figure 7. Bonus features

in the gaps, where LSP is lacking, in order for our solution to function properly. Visual Studio Code allows plug-in developers to hook into specific events caused by user interaction, in accordance to the LSP standard. Our solution is built upon these event hooks, allowing the communication with the code analyzer whenever the developer interacts with their code or their editor. This lets us update the call graph in real-time, reflecting the changes which the user has made.

The client uses the aforementioned event hooks, in order to detect when the user is interacting with the editor. This information is then passed along to the server, where the source code is processed, and a call graph is then generated and displayed inside of the code editor. See Figure 3 for a visualization.

3 Evaluation

We tested the code on both a simple example with inherited classes and functions, and then a complex project. The extension has a scalability limit. It is easy to generate large graphs as more files are added to the list of dependencies (see Figure 8). This means that in practice there is an upper limit to how many files can be analyzed at a time before the call graph is too messy to read. The list of dependencies is easy to add to and remove from, which allows the user to switch out files when the graph reaches an unintelligible level of node and edges. The different depth level feature may also be used to get a higher overview of how files interact with fewer nodes, at the cost of detailed information.

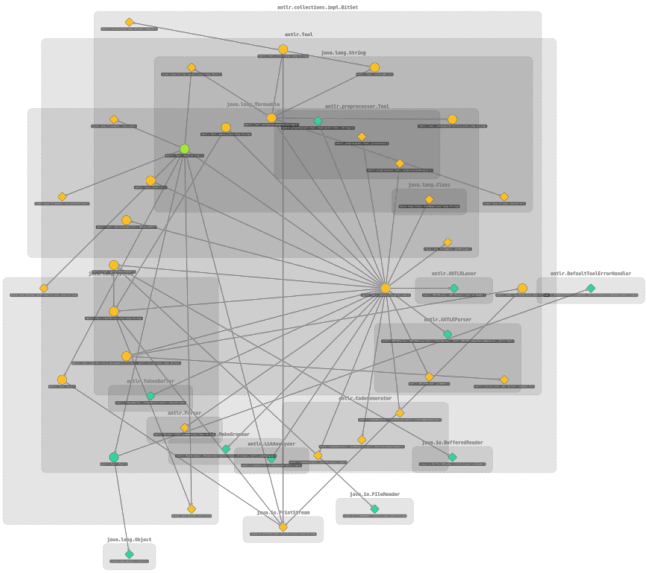


Figure 8. Extension on complex project

3.1 Limitations

We focused specifically on the Java programming language and our tool therefore only supports Java code currently,

although the client of our solution could technically be reworked to fit any object-oriented programming language. The server is language specific, but it may be switched out for a different language server if the data it returns follows the JSON format accepted by the client.

We do not generate graphs from bytecode in our solution, since that would require a decompiler to be implemented in order to handle the task, which was considered out-of-scope for this project. The drawback of this is that methods that are contained within compiled `.class` or `.jar` files cannot be expanded.

3.2 Future work

In our current solution we need to manually choose an entry point. It would be pleasant to have buttons allowing the call graph to start with an entry point from where we are in the code, as visualized in Figure 9.

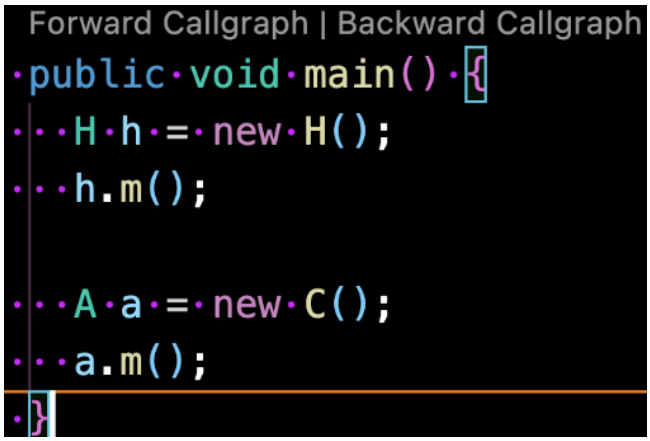


Figure 9. Starting call graph from within code

In our current implementation, we can highlight a forward call graph from a specific node. A forward call graph entails all nodes reachable from that node. As implemented in Figure 9, our tool could be further developed to show a backward call graph. A backward call graph would instead show all nodes that the selected node is reachable from. A backward call graph could be misleading, as there may be files not in our list of dependencies that call the current node. We may think a node is never called and therefore dead code, when in reality we have not analyzed the files where the code is used. The only reliable way to find dead code would be to analyze the whole codebase, which could lead to serious performance and readability issues if the codebase is large enough.

An improvement with no drawbacks that we did not implement would be to have the server start automatically with the client, rather than forcing the user to start the server manually.

As mentioned in our introduction, the tool could be developed to be a visual debugging tool, which would step through each node inside the graph together with the usual debugger.

4 Related work

A similar project to ours is Crabviz [7]. Crabviz is, just as our approach, an LSP-based call graph generator. It is available for Visual Studio Code, and offers a variety of functionality, such as highlighting, support for multiple languages, exporting graphs as SVGs, etc. The call graph is visually different from our extension, with nodes that are text boxes similar to that of a UML diagram, rather than the simple geometric shapes used in ours.

CodeProber with IntraJ is a tool for visualizing control-flow graphs on top of source code [1]. As in our approach, the tool utilizes RAGs and ASTs, and is a visual tool for program analysis.

Our project took some visual inspiration, such as having multiple depth levels and expandable nodes, from CodeScene, a software analysis tool used to help visualize contributions and code quality in software projects [3].

5 Conclusion

In this paper, we addressed the need for visual program analysis tools within development environments. By using the LSP and a client-server architecture with the server as a language server, we presented a call graph visualization tool integrated into Visual Studio Code. Our tool allows developers to analyze parts of a Java codebase, leading to better understanding of the code. With its many features, the tool provides a flexible call graph. Our contribution is a demonstration of how tools integrated with development environments can provide support to developers.

It is limited in language support, only being developed for Java, and in scalability for larger codebases. Future work includes extending support to other programming languages, backward call graph generation, and integration with debuggers for even deeper insights.

References

[1] Anton Risberg Alaküla. 2024. CodeProber - Source code based exploration of program analysis results. https://github.com/lu-cs-sde/codeprober. (2024).

[2] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: An empirical study. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 332–343.

[3] CodeScene. [n. d.]. CodeScene. https://codescene.com/. ([n. d.]).

[4] Max Franz, Christian T. Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D. Bader. 2015. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics* 32, 2 (09 2015), 309–311. https://doi.org/10.1093/bioinformatics/btv557 arXiv:https://academic.oup.com/bioinformatics/article-pdf/32/2/309/49016536/bioinformatics_32_2_309.pdf

[5] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica* 24, 3 (2000), 301–317.

[6] Görel Hedin. 2011. Tutorial: An Introductory Tutorial on JastAdd Attribute Grammars. In *GTTSE III (Lecture Notes in Computer Science)*, Vol. 6491. 166–200. https://doi.org/10.1007/978-3-642-18023-1_4
[7] Chan HoCheung. 2024. Crabviz. https://github.com/chanhx/crabviz. (2024).
[8] Microsoft. 2024. Language Server Protocol. https://microsoft.github.io/language-server-protocol/. (2024).
[9] ExtendJ Project. [n. d.]. ExtendJ: The Extensible Java Compiler. https://extendj.org/. ([n. d.]). Accessed: 2024-12-23.
[10] Idriss Riouak. 2024. CAT - CallGraph Analysis Tool. https://github.com/IdrissRio/cat. (2024).
[11] Idriss Riouak, Niklas Fors, Jesper Öqvist, Görel Hedin, and Christoph Reichenbach. 2024. Efficient Demand Evaluation of Fixed-Point Attributes using Static Analysis. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24)*. Association for Computing Machinery, New York, NY, USA, 56–69. https://doi.org/10.1145/3687997.3695644
[12] B.G. Ryder. 1979. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering* SE-5, 3 (1979), 216–226. https://doi.org/10.1109/TSE.1979.234183