Djonathan Douglas Santos Barros

Suporte à Edição para Linguagens de Software: Práticas de Implementação com *Language Server Protocol*

Cascavel-PR

2022

Djonathan Douglas Santos Barros

Suporte à Edição para Linguagens de Software: Práticas de Implementação com *Language Server Protocol*

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-Graduação em Ciência da Computação (PPGComp) da Universidade Estadual do Oeste do Paraná – Unioeste, campus de Cascavel.

Universidade Estadual do Oeste do Paraná – Unioeste – Cascavel

Centro de Ciências Exatas e Tecnológicas – CCET

Programa de Pós-Graduação em Ciência da Computação – PPGComp

Orientador: Dr. Wesley Klewerton Guêz Assunção

Coorientador: Dr. Thorsten Berger

Cascavel-PR

2022

Djonathan Douglas Santos Barros

Suporte à Edição para Linguagens de Software: Práticas de Implementação com *Language Server Protocol*/ Djonathan Douglas Santos Barros. – Cascavel-PR, 2022-

97p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Wesley Klewerton Guêz Assunção

Dissertação (Mestrado)– Universidade Estadual do Oeste do Paraná – Unioeste – Cascavel

Centro de Ciências Exatas e Tecnológicas – CCET

Programa de Pós-Graduação em Ciência da Computação – PPGComp, 2022.

1. Language engineering. 2. Code assistance. 3. Source Code Editor 4. Implementation practices. I. Wesley Klewerton Guez Assunção. II. Universidade Estadual do Oeste do Paraná – Unioeste – Cascavel. III. Programa de Pós-Graduação em Ciência da Computação - PPGComp. IV. Suporte à Edição para Linguagens de Software: Práticas de Implementação com *Language Server Protocol*

Djonathan Douglas Santos Barros

Suporte à Edição para Linguagens de Software: Práticas de Implementação com *Language Server Protocol*

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-Graduação em Ciência da Computação (PPGComp) da Universidade Estadual do Oeste do Paraná – Unioeste, campus de Cascavel.

Trabalho aprovado. Cascavel-PR, 05 de dezembro de 2022:

Dr. Wesley Klewerton Guêz Assunção
Orientador

Dr. Ivonei Freitas da Silva
Avaliador Interno

Dr. Fernando José Castor de Lima Filho
Avaliador Externo

Cascavel-PR
2022

Este trabalho é dedicado aos meus pais,
que me ensinaram o valor da educação e do aprendizado.

Agradecimentos

Agradeço a Deus, que me abençoa a cada dia com saúde para continuar estudando e trabalhando, que cruzou o meu caminho com o da minha esposa para construirmos nossa família e que me mantém firme diante das adversidades da vida.

Agradecimentos especiais a Unioeste, que organizou este programa de pós graduação no Oeste do Paraná, gerando oportunidades para os profissionais da região continuarem se aprimorando e promovendo o desenvolvimento do nosso ecossistema.

Ao meu orientador Wesley Klewerton Guez Assunção que me guiou durante esses anos, tirou minhas dúvidas, aguentou minhas reclamações e sempre me estimulou a pensar além do trivial.

Ao Thorsten Berger e Sven Peldszus que nos auxiliaram com ideias e inspirações para o trabalho. A experiência desses profissionais em engenharia de linguagens foi crucial durante o refinamento da análise dos resultados.

Agradeço também a minha esposa, bem como a minha família, que me deram o suporte necessário durante meus momentos de estresse, que me estimularam a ir até o fim e que compreenderam os meus momentos de ausência.

“"Mestre, qual é o maior mandamento da Lei?",
Respondeu Jesus: "'Ame o Senhor, o seu Deus de todo o seu coração,
de toda a sua alma e de todo o seu entendimento'.,
Este é o primeiro e maior mandamento.
E o segundo é semelhante a ele: 'Ame o seu próximo como a si mesmo'.
Deste dois mandamentos dependem toda a Lei e os profetas."
(Bíblia Sagrada - Tradução NVI, Mateus 22, 36-40)

Resumo

BARROS, Djonathan Douglas Santos. **Suporte à Edição para Linguagens de Software: Práticas de Implementação com *Language Server Protocol***. Orientador: Dr. Wesley Klewerton Guêz Assunção. 2022. 97f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual do Oeste do Paraná, Cascavel – Paraná, 2022.

A utilização efetiva de linguagens de software, sejam elas de programação ou linguagens especificas de domínio, requer um suporte à edição efetivo. IDEs modernas, ferramentas de modelagem, editores de código tipicamente disponibilizam um suporte à edição sofisticado para criar, compreender ou modificar instâncias (programas ou modelos) de linguagens em particular. Infelizmente, implementar tal suporte é desafiador. Enquanto a engenharia de linguagens é uma disciplina bem conhecida e suportada por modernas técnicas dirigidas por modelo, há uma escassez de princípios e práticas para implementar o suporte a edição. Especialmente para linguagens especificas de domínio (que são geralmente criadas por organizações menores ou desenvolvedores individuais, algumas vezes somente para projetos únicos) se beneficiariam de melhores métodos e ferramentas para criar um suporte à edição apropriado. Nesse trabalho foram identificados aspectos de implementação para o suporte a edição em 30 servidores de linguagem que implementam o *language server protocol* (LSP), alguns desenvolvidos para suportar múltiplas linguagens. O LSP é um padrão consumado para implementar o suporte à edição para uma linguagem, separado das ferramentas de edição (ex., IDEs ou ferramenta de modelagem), aumentando o reúso e a qualidade do suporte a edição. Evidenciando a popularidade do LSP, existem 125 servidores de linguagem catalogados pela comunidade hoje em dia. Além dos aspectos de implementação que os desenvolvedores devem levar em consideração ao implementar o suporte à edição, foram sintetizadas práticas de implementação para endereça-los, baseado em uma análise sistemática do código-fonte desses servidores.

Palavras-chave: engenharia de linguagem; assistência de código; compreensão de código; editor de código-fonte; práticas de implementação

Abstract

BARROS, Djonathan Douglas Santos. **Editing Support for Software Languages: Implementation Practices in Language Server Protocols**. Orientador: Dr. Wesley Klewerton Guêz Assunção. 2022. 97f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual do Oeste do Paraná, Cascavel – Paraná, 2022.

Effectively using software languages, be it programming or domain-specific languages, requires effective editing support. Modern IDEs, modeling tools, and code editors typically provide sophisticated support to create, comprehend, or modify instances (programs or models) of particular languages. Unfortunately, building such editing support is challenging. While the engineering of languages is well understood and supported by modern model-driven techniques, there is a lack of engineering principles and best practices for realizing their editing support. Especially domain-specific languages (often created by smaller organizations or individual developers, sometimes even for single projects) would benefit from better methods and tools to create proper editing support. We study aspects for implementing editing support in 30 language servers—implementations of the language server protocol (LSP), some of which support multiple languages. The LSP is a recent de facto standard to realize editing support for languages, separated from the editing tools (e.g., IDEs or modeling tools), enhancing the reusability and quality of the editing support. Witnessing the LSP's popularity there are 125 language servers catalogated in a curated list. We identify concerns that developers need to take into account when developing editing support, and we synthesize implementation practices to address them, based on a systematic analysis of the servers' source code.

Keywords: language engineering; code assistance; code comprehension; source code editor; implementation practices

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AST	Abstract Syntax Tree
DSL	Domain Specific Language
DOM	Document Object Model
IDE	Integrated Development Environment
GPL	General Purpose Language
LSP	Language Server Protocol
RPC	Remote Procedure Call
SDK	Software Development Kit
URI	Uniform Resource Identifier

Sumário

1 Introdução

Linguagens de software são cruciais não somente para a engenharia de software, mas também para várias outras disciplinas de engenharia que necessitem da criação de modelos e automação de tarefas (LÄMMEL, 2018). Utilizar efetivamente linguagens específicas de domínio, de propósito geral ou de modelagem (COOK et al., 2017), requer um efetivo suporte à edição. IDEs modernas e ferramentas de modelagem frequentemente são embarcadas com um sofisticado suporte para criar, compreender, e modificar modelos ou programas escritos em determinada linguagem (ERDWEG et al., 2013). Funcionalidades tipicas de suporte à edição incluem: *sugestões de código*, *destaque de sintaxe*, *marcação de erros*, *formatação*, *refatorações*, dentre outras.

Infelizmente, criar um suporte à edição apropriado para linguagens é uma atividade complexa. Tendo em vista as grandes comunidades ou mesmo grandes corporações mantendo as linguagens populares, espera-se que os criadores de ferramentas de engenharia de software (ou mesmo de ferramentas de modelagem) se beneficiem desses recursos, fornecendo o suporte à edição necessário. Enquanto, para linguagens de modelagem específicas de domínio, que são frequentemente criadas por organizações menores ou desenvolvedores individuais e tem seus usos limitados muitas vezes a alguns poucos projetos, esses recursos tendem a ser mais escassos. Ao mesmo tempo, tais linguagens com menos recursos cumprem um papel de destaque possibilitando a automação da engenharia de software (BIALY et al., 2017; WASOWSKI; BERGER, 2022) e oferecendo notações semanticamente ricas e concisas (BÜNDER, 2019; DRAGULE et al., 2021). Em ambos os casos, as linguagens se beneficiariam ao obter suporte à edição, permitindo que seus usuários foquem em resolver problemas reais, ao invés de gastar tempo aprendendo sua sintaxe.

1.1 Motivação

Pesquisadores e profissionais vêm trabalhando intensamente em métodos e ferramentas para a construção de linguagens (WASOWSKI; BERGER, 2022; FOWLER, 2010; AHO et al., 2007; PARR, 2009). Graças às modernas técnicas de engenharia de linguagens dirigida por modelos (ex., meta-modelagem, mapeamento automático de sintaxe abstrata para concreta, e transformações de modelos) que são frequentemente integradas à *language workbenches* (ferramentas para criação de DSLs) modernos e convenientes (ERDWEG et al., 2013), as arquiteturas para infraestrutura de linguagens (LÄMMEL, 2018; VÖLTER et al., 2013) e os processos de engenharia de linguagem são bem entendidos (WASOWSKI; BERGER, 2022; FOWLER, 2010). Além de toda teoria e tecnologia em engenharia de linguagens, existem padrões de implementação para linguagens de programação (PARR,

2009; AHO et al., 2007). Por exemplo, existem recomendações de como construir árvores abstratas de sintaxe (ASTs), como implementar navegação/reescrita de árvores ou como modularizar arquiteturas de linguagem (BASTEN et al., 2015). Especificamente para linguagens específicas de domínio, padrões recorrente para análise de domínio, design e implementação já foram reportados (SPINELLIS, 2001; MERNIK; HEERING; SLOANE, 2005). Porém, há uma falta de princípios de engenharia e melhores práticas para a implementação do suporte à edição para essas mesmas linguagens.

O *Language Server Protocol (LSP)* (MICROSOFT, 2022b; GUNASINGHE; MARCUS, 2022) é uma iniciativa recente, de 2016, para modularizar o suporte à edição em assim chamados servidores de linguagem. Sua abordagem é focada no fato do suporte à edição, implementado especificamente para uma linguagem de software, ser dependente e específico para cada IDE ou editor de texto, impossibilitando seu reúso ou extensão em diferentes ferramentas. O LSP permite que os engenheiros tornem suas linguagens, sejam de programação ou específicas de domínio, disponíveis em uma grande variedade de ferramentas de edição, com um mínimo esforço para adoção e reúso. O LSP descreve uma API comum que pode ser implementada por desenvolvedores de linguagens e reutilizada por diferentes clientes (editores e IDEs) (MICROSOFT, 2022b). As ferramentas de edição se conectam ao servidor de linguagem, onde funcionalidades individuais de suporte à edição podem ser requisitadas para determinado modelo ou programa. O LSP utiliza a terminologia *Documento* para definir as instâncias de linguagens (ex., um arquivo de código-fonte), termo que será utilizado no decorrer do trabalho. Originalmente o LSP foi proposto pela Microsoft para prover suporte a múltiplas linguagens para o editor de texto Visual Studio Code (MICROSOFT, 2022b). Um total de 23 funcionalidades são definidas pelo padrão até o momento, excluindo métodos auxiliares para executar etapas dessas funcionalidades ou controlar o ciclo de vida do servidor. Em Dezembro/2022, 125 diferentes servidores de linguagens são listados em uma lista curada de implementações dirigida pela comunidade (SOURCEGRAPH, 2022).

1.2 Problema

Apesar de ser uma alternativa para amenizar o problema de portabilidade de IDEs, o LSP se esforça apenas em definir: (i) os aspectos técnicos do protocolo de requisição e resposta que devem ser utilizados tanto pelo cliente quanto pelo servidor; e (ii) esquemas de mensagem para cada funcionalidade de edição. Portanto, fica a critério do mantenedor do servidor de linguagem definir, quais funcionalidades implementar, bem como os algoritmos de implementação. Engenheiros que não tenham experiência na implementação de suporte à edição para linguagens de software ou na implementação de servidores LSP se beneficiariam de um conjunto de práticas ou diretrizes que os guiassem durante o desenvolvimento.

O conjunto de servidores catalogados pela comunidade (SOURCEGRAPH, 2022) deve lidar com uma série de aspectos para implementar o suporte à edição para cada linguagem que eles suportam. Princípios recorrentes de engenharia podem estar contidos nesses servidores de linguagem, pavimentando o caminho para nosso objetivo de longo prazo de estabelecer catálogos de padrões para a implementação do suporte à edição. Dados esses aspectos esperados, diferentes servidores devem fornecer abordagens diferentes para atendê-los. Catalogar tanto os aspectos de implementação, quanto o conjunto de práticas existentes em servidores LSP existentes, pode servir como um aconselhamento prático aos engenheiros de linguagens e pesquisadores sobre como implementar o suporte à edição de forma efetiva ou estruturar um servidor de linguagem.

1.3 Objetivos

A partir do problema formulado, o presente trabalho tem como objetivo **identificar quais são os aspectos de implementação e práticas utilizadas para implementar o suporte à edição para linguagens de software em servidores LSP existentes**. Objetivando identificar detalhes sobre: (i) como arquitetar o suporte à edição, desde a definição de funcionalidades a serem implementadas até na estruturação e design do projeto; e (ii) como implementar efetivamente as funcionalidades de suporte à edição. O objetivo geral será abordado a partir dos seguintes objetivos específicos:

- Compreender como os servidores LSP são implementados na prática
- Identificar se existem aspectos práticos que guiam a arquitetura e implementação do suporte à edição
- Reportar as práticas utilizadas pelos desenvolvedores de servidores atuais para atender esses aspectos

Espera-se proporcionar aos pesquisadores uma compreensão sobre os esforços necessários para a implementação prática de suporte à edição para linguagens de software, incitando assim uma discussão sobre melhores práticas e eventualmente o desenvolvimento de uma teoria e de técnicas originais para a efetiva implementação desse suporte. Também que os profissionais possam utilizar nossos achados como uma base para novos projetos ou mesmo na otimização de implementações existentes, utilizando nosso conjunto de aspectos e práticas como diretrizes ou inspiração.

1.4 Organização do Trabalho

O restante do trabalho está organizado da seguinte maneira: O Capítulo 2 apresenta uma introdução aos principais conceitos que embasam o nosso trabalho como conceitos de

Engenharia de Linguagens, detalhes sobre o funcionamento do *Language Server Protocol* e uma breve comparação entre esse trabalho e outros relacionados. O Capítulo 3 apresenta a metodologia utilizada para selecionar e analisar a implementação dos servidores. O Capítulo 4 apresenta os resultados da análise realizada através de um catálogo de aspectos e práticas. O Capítulo 5 contém a discussão sobre os achados. O Capítulo 6 apresenta desafios à validade do trabalho, tanto interna quanto externa. O Capítulo 7 apresenta uma discussão final sumarizando as contribuições (tanto para a indústria quanto para a acadêmia) bem como propostas de trabalhos futuros.

2 Referencial Teórico

Neste capítulo, são apresentados os conceitos básicos referentes a linguagens de software, suporte à edição e uma visão geral do funcionamento do LSP. Os conceitos estão diretamente relacionados com os resultados apresentados, formando uma base de conhecimento para compreendê-los.

2.1 Suporte à edição para linguagens de software

A implementação de um software é a etapa do desenvolvimento onde uma Linguagem de programação de *Propósito Geral* (GPL) ou *Específica de Domínio* (DSL) é utilizada para automatizar a solução de um problema de forma algorítmica (LÄMMEL, 2018). Durante a implementação, os desenvolvedores estendem a gramática básica da linguagem através da declaração de estruturas de dados reutilizáveis (ex., classes, registros, ou sub-rotinas), bem como definem símbolos para identificar as variáveis, constantes, e outras instâncias do programa/modelo que está sendo implementado (SEBESTA, 2004). Conhecer a gramática básica da linguagem de programação utilizada bem como o conjunto de símbolos declarados especificamente para o projeto requer esforço por parte dos desenvolvedores, que são beneficiados diretamente por editores que forneçam recursos que facilitem essa compreensão (FEKETE; PORKOLÁB, 2020).

Xia et al. (2017) avaliou que um profissional gasta em média 58% de seu tempo em atividades que visam a compreensão do programa em que está trabalhando, 24% navegando pelo código, e apenas 5% o editando. A experiência do desenvolvedor, bem como a atividade que está sendo realizada (ex., manutenção), são fatores que influenciam no tempo que será gasto para a compreensão do código. No mesmo trabalho foi observado que os profissionais passaram 20% do tempo de compreensão navegando pelo projeto utilizando seus ambientes integrados de desenvolvimento (IDEs). Além disso, observou que 10% do tempo foi desprendido em editores de texto, ambas ferramentas que fornecem funcionalidades automatizadas para compreender o código-fonte. O suporte à edição para linguagens de software deve então auxiliar os desenvolvedores durante a escrita, compreensão, e modificação de documentos. Sendo que esses documentos, em mais alto-nível, são instâncias de linguagens como modelos ou programas (código-fonte).

Em um trabalho recente, Mészáros, Cserép e Fekete (2019) avaliaram quais funcionalidades de compreensão de código (*Code Comprehension*) são oferecidas por IDEs populares. Por meio dessa pesquisa, os autores demonstram que funções que auxiliem na navegação pelo projeto como `Goto Definition`, `Find All references` e `Symbol Search` são as mais populares entre as ferramentas avaliadas. O mesmo estudo também relata a alta

popularidade da funcionalidade `Code Completion`, que além de auxiliar na compreensão do programa, provendo ao usuário recomendações de símbolos que podem ser utilizados, também serve como uma referência ao sugerir fragmentos para criar código a partir de modelos pré estabelecidos (`code snippets`).

Além da complexidade de implementar cada uma das funcionalidades de suporte à edição, os desenvolvedores podem preferir utilizar diferentes ferramentas de edição, como diferentes IDEs ou editores. Isso significa que uma linguagem em particular deve ser suportada por uma variedade de ferramentas (RODRIGUEZ-ECHEVERRIA et al., 2018). De outra perspectiva, para os mantenedores dessas ferramentas, é também um grande esforço prover suporte para múltiplas linguagens, pois além de desenvolver tal suporte, eles devem ainda acompanhar a evolução das novas versões das linguagens.

Esse comportamento pode ser representado pela equação $M * N$, onde M é a quantidade de linguagens a serem suportadas e N representa a quantidade de IDEs ou editores que devem ser suportadas. Por exemplo, para suportar as 10 linguagens mais populares em três ferramentas de edição (ex., Eclipse IDE, Jetbrains Fleet, Visual Studio Code) serão necessárias 30 implementações diferentes. Essa relação é chamada de **problema da portabilidade de IDEs**.

Um exemplo de iniciativa para reduzir o esforço necessário para reutilizar a implementação do suporte à edição já criado para uma linguagem em diferentes ferramentas de edição é o LSP. O protocolo deve ser seguido tanto pelas ferramenta quanto pelas implementações do suporte para cada linguagem.

2.2 O Language Server Protocol

O LSP (MICROSOFT, 2022b) estabelece um protocolo cliente-servidor que possibilita os clientes (ex., as ferramentas de edição) a requisitarem o suporte à edição provido pelos servidores de linguagem, ambos em diferentes processos do computador. A especificação LSP estabelece quais são as funcionalidades básicas que podem ser suportadas pelo servidor, bem como os esquemas das mensagens que serão trocadas.

O suporte típico, chamado de *language smarts* na terminologia do LSP (BÜNDER, 2019), incluí renomear, navegar, ou exibir documentações ao posicionar o mouse sobre um símbolo. As funcionalidades que podem ser implementadas são definidas pela especificação do protocolo (MICROSOFT, 2022b). Contudo, cada servidor LSP pode implementar diferentes funcionalidades dependendo das características da linguagem ou o suporte à edição desejado.

2.2.1 Funcionalidades do LSP

O suporte à edição é encapsulado através de métodos individuais, representando um total de 23 funcionalidades atualmente especificadas pelo LSP. O servidor de linguagem tem, assim como seus clientes, acesso ao assim nomeado espaço de trabalho (*workspace*). O espaço de trabalho contém os documentos para os quais o suporte à edição é necessário. Outra abstração, chamada de símbolos (*symbols*) pelo padrão LSP, se refere a qualquer elemento textual da linguagem para o qual algum suporte pode ser disponibilizado, como identificadores, palavras-chave, expressões ou qualquer outro conceito estrutural (ex., classes, métodos ou constantes).

Tabela 1 – Sumário de funcionalidades em ordem descendente

Feature	**Categoria**	**Versão**
Goto Definition	COMPR	
Completion	ASSIST	
Hover	COMPR	
Diagnostics	ASSIST	
Find References	COMPR	
Document Symbols	COMPR	
Signature Help	COMPR	
Document Formatting	ASSIST	
Folding Range	COMPR	3.10.0
Workspace Symbols	COMPR	
Rename	ASSIST	
Code Action	AUX	
Execute Command	AUX	
Document Highlights	COMPR	
Code Lens	COMPR	
Document Range Formatting	ASSIST	
Goto Type Definition	COMPR	3.6.0
Goto Implementation	COMPR	3.6.0
Document on Type Formatting	ASSIST	
Document Link	COMPR	
Selection Range	ASSIST	3.15.0
Document Color	COMPR	3.6.0
Goto Declaration	COMPR	3.14.0

ASSIST = Code Assistance, COMPR = Code Comprehension, AUX = Auxiliar

A Tabela 1 apresenta uma relação de funcionalidades de edição especificadas pelo protocolo. As funcionalidades mais populares serão detalhadas no Capítulo 4.2. Baseado na descrição das funcionalidades retiradas do texto da especificação (MICROSOFT, 2022b), pode-se classificar as funcionalidades em três diferentes categorias.

- *Code assistance:* Essa categoria engloba 7 funcionalidades que auxiliam os desenvolvedores ao escrever novo código. Essas funcionalidades geralmente sugerem modificações

no documento, como renomear símbolos, formatar o código ou sugerindo opções para completar um determinado prefixo. A funcionalidade `Diagnostics` foi classificada nessa categoria, pois além da mensagem de diagnóstico, o servidor pode sugerir alterações para corrigir o problema (*quick fix*).

- *Code comprehension:* Essa categoria engloba 14 funcionalidades que auxiliam os desenvolvedores ao explorar o código-fonte. Essas funcionalidades geralmente não modificam os documentos, mas auxiliam os desenvolvedores ao prover documentação e navegação dentro ou entre diferentes documentos.

- *Auxiliar:* Essa categoria compreende 2 funcionalidades que permitem que as implementações de servidores LSP disponibilizem habilidades adicionais, não especificadas previamente pelo padrão ou executar etapas adicionais de uma funcionalidade específica. Um exemplo é a funcionalidade `Execute Command`, que permite que os servidores implementem habilidades adicionais, como refatorações.

2.2.2 Esquema de Requisições e Respostas do LSP

A comunicação parte do cliente que define o momento em que o servidor será iniciado. O protocolo que rege os aspectos técnicos dessa interação é o JSON-RPC[1], estabelecendo que em uma chamada de procedimento remota para o servidor, tanto a funcionalidade requisitada quanto os parâmetros para a execução serão informados no corpo da requisição. Para trafegar as informações é utilizada uma representação das mensagens em formato JSON. O servidor é considerado *Stateful*, pois necessita de acesso aos arquivos com os códigos-fonte do projeto (*workspace*). A cada requisição, o cliente deve informar todas as informações necessárias para o servidor executar a operação, como: (i) qual o arquivo que será operado, (ii) posicionamento do cursor, e (iii) parâmetros adicionais definidos para cada funcionalidade. A partir dessas informações o servidor carrega o conteúdo do arquivo e realiza a operação solicitada, retornando o resultado também em formato JSON.

Na Figura 1 é apresentado um exemplo[2] de comunicação entre o Cliente (*Development Tool*) e o Servidor de Linguagem (*Language Server*). Nesse exemplo o cliente notifica que está com um documento em edição e a cada modificação o servidor enviará um diagnóstico sobre o conteúdo do arquivo (ex., resposta de um *Linter* ou Compilador com erros/avisos relacionados a sintaxe). O cliente então solicita a execução da funcionalidade `Goto Definition`, recebendo do servidor a localização (DocumentURI, linha e coluna) onde um determinado símbolo foi definido. O resultado então, pode ser utilizado pelo cliente para navegação. Ao final do processo o cliente notifica ao servidor que não está

[1] <https://www.jsonrpc.org/>
[2] Extraído de: <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>

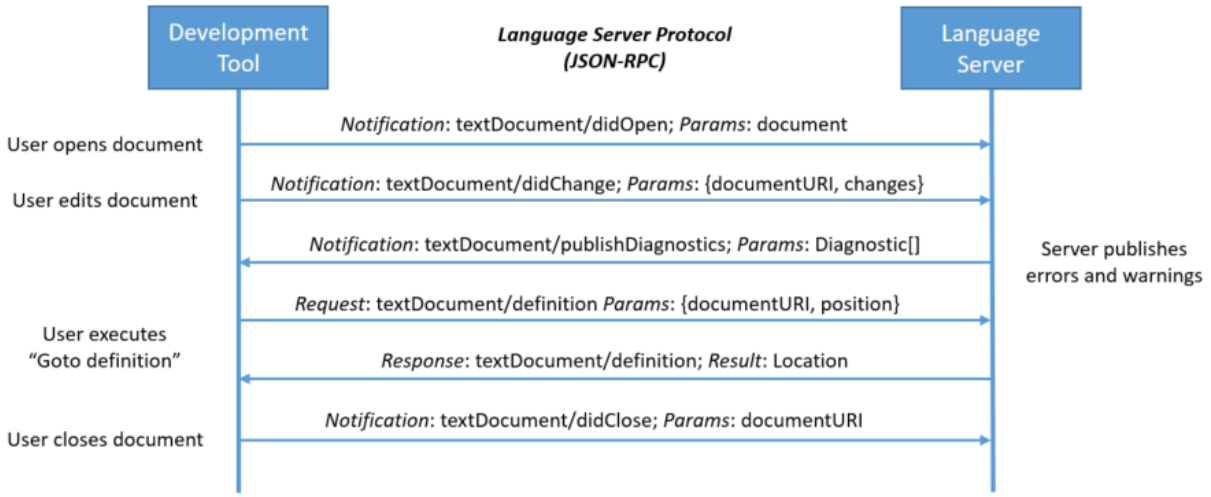

Figura 1 – Exemplo de comunicação utilizando o *Language Server Protocol*

mais com o arquivo aberto, não sendo necessária a notificação de novas mensagens de diagnósticos.

O Código de exemplo da Figura 2 ilustra a mensagem enviada pelo cliente para requisitar a funcionalidade *textDocument/definition*. O servidor munido dessas informações pode então carregar os arquivos necessário no *workspace* e identificar onde o símbolo solicitado foi declarado, retornando essa localização (URI do documento e intervalo onde está declarado) também via JSON-RPC (Figura 3).

```
1  {
2      "jsonrpc": "2.0",
3      "id": 1,
4      "method": "textDocument/definition",
5      "params": {
6          "textDocument": {
7              "uri": "file:///src/main/java/Main.java"
8          },
9          "position": {
10             "line": 30,
11             "character": 80
12         }
13     }
14 }
```

Figura 2 – Exemplo de requisição via JSON-RPC da feature `Goto definition`

Os aspectos apresentados nessa seção demonstram quais são as mensagens, bem como o funcionamento do protocolo de comunicação entre o cliente e servidor, de acordo com o estabelecido pelo LSP. A seguir, serão apresentados detalhes de como o código-fonte do projeto pode ser representado e navegado para a implementação das funcionalidades.

Figura 3 – Exemplo de resposta via JSON-RPC da feature `Goto definition`

2.3 Fundamentos de Engenharia de Linguagem para a implementação do suporte à edição

A implementação de um suporte à edição efetivo requer o devido conhecimento das especificidades de implementação de cada linguagem, que é normalmente composta de símbolos que podem representar palavras-chave ou mesmo operações aritméticas e operações especiais de linguagem (ex., ponteiros ou *arrow functions*) (GUNASINGHE; MARCUS, 2022; LÄMMEL, 2018). Além disso, durante a criação do programa símbolos adicionais são criados pelo programador para representar diferentes instâncias como variáveis, classes, constantes, dentre outras. As funcionalidades de suporte tem a responsabilidade de identificar símbolos e navegar entre eles, necessitando assim de uma representação estruturada e não textual dos documentos.

Apesar de não trivial, a criação de estruturas de dados que representem de forma navegável um programa já se faz presente em ferramentas que suportam a criação dessas linguagens. Tanto as GPLs quanto DSLs necessitam de ferramentas para transformar a representação da instância código-fonte/modelo em código executável. Esse processo pode ser implementado por compiladores que tratam de todas as fases dessa transformação. A estrutura de um compilador é composta de (i) um *front-end*, abrangendo ferramentas que transformam o código-fonte em uma instância que pode ser interpretada e otimizada e (ii) um *back-end*, abarcando ferramentas geram instruções de máquina a partir de uma instância intermediária (COOPER; TORCZON, 2011).

As ferramentas de *front-end*, fornecem o necessário para que os engenheiros que implementam o suporte à edição consigam identificar os símbolos e a estrutura geral de um programa, e assim coletar as informações necessárias para implementar suas funcionalidades.

Dentre essas ferramentas, podemos citar (i) *Scanners*, responsáveis pela análise léxica do código-fonte, inspecionando seu conteúdo como um fluxo de caracteres e transformando o mesmo em um fluxo de símbolos (*tokens*); e os (ii) *Parsers*, que fazem a análise sintática e semântica dos símbolos identificados, gerando assim uma árvore de sintaxe com a estrutura do documento (AHO et al., 2007).

Um exemplo de ferramenta de *front-end* para geração de linguagens de software, é a utilização de formalismos (ex., gramáticas livres de contexto). Onde, uma vez definida a gramática de linguagem, ela pode então ser utilizada por geradores de *parsers* (ex., ANTLR (PARR, 2022) e XText (COMMUNITY, 2022)). O ANTLR é utilizado por várias GPLs, que a partir dos símbolos definidos por gramáticas, gera um *parser* específico para a linguagem suportada. Além de ferramentas baseadas em *parsing*, DSLs comumente se beneficiam de ferramentas que simplificam a criação dessas linguagens (WASOWSKI; BERGER, 2022). Um exemplo é a utilização de *Language Workbenches*, que geram editores e interpretadores para a linguagem a partir de Células de Gramática, ambos integrados e específicos para determinada ferramenta de edição.

Vários são os aspectos que tornam a implementação do suporte à edição uma atividade de engenharia não trivial. Primeiro, a escolha de uma ferramenta/biblioteca de *parsing*. Então, o mapeamento das palavras-chave da linguagem a ser suportada. Além da coleta de informações de tempo de execução e de informações do ambiente de desenvolvimento. Como atividade opcional, suportada por padrão nos SDKs de algumas linguagens (ex., Golang), a formatação apropriada do código-fonte.

Representação intermediária do código: Para executar as funcionalidades, é necessário uma estratégia para coletar informações sobre os símbolos presentes no código-fonte (ex., tipo de símbolo e tipo de dados) (AHO et al., 2007; COOPER; TORCZON, 2011). Explorar a sintaxe concreta, via algoritmos de manipulação de texto ou Expressões Regulares pode ser custoso e mesmo assim não fornecer todos os detalhes necessários para a eficiente execução das tarefas.

O processo de compilação de instâncias de texto para código intermediário, ou código de máquina, requer várias etapas. A primeira é a identificação de *tokens* a partir dos separadores estabelecidos na gramática da linguagem. Uma vez que os *tokens* são extraídos e sintaticamente identificados/rotulados, a relação semântica entre eles é validada, e estruturas de dados podem ser utilizadas para representar essas relações. Essas estruturas são utilizadas com diferentes intuitos, desde a análise estática de código-fonte, até a geração de diagramas ou otimizações (COOPER; TORCZON, 2011).

Uma abordagem comum para a navegação nas estruturas de um código-fonte é através de uma representação abstrata do código em *Document Object Model* (DOM) ou formalmente *Abstract Syntax Model* (AST) (AHO et al., 2007). Um exemplo simplificado de uma AST pode ser encontrado na Figura 4, onde uma simples instrução para imprimir

o resultado de uma expressão aritmética implementada em Java é demonstrada. Cada nó da árvore representa uma estrutura lógica da instrução, as arestas simbolizam as relações entre essas estruturas. Diferentes compiladores/*parsers* podem criar representações com mais ou menos detalhes dos nós (ex., *tipo do nó, rótulo, localização no código-fonte, e local em que o símbolo foi declarado*), bem como cada nó pode ser representado por uma classe/*struct* com seus próprios atributos (ex., as expressões binárias da figura contém um atributo indicando o operador utilizado).

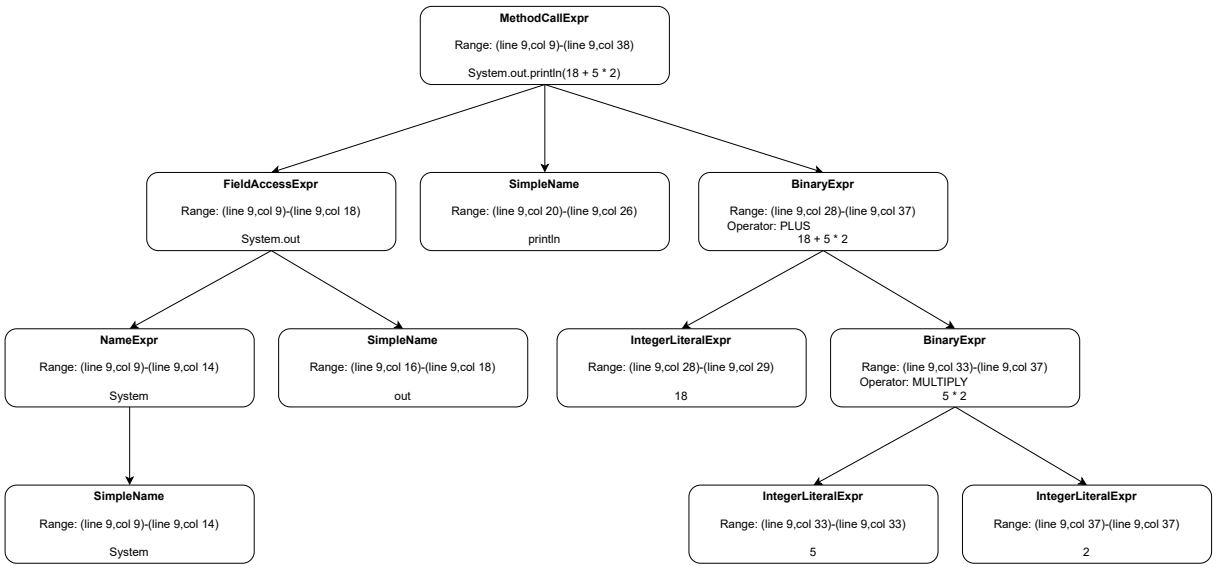

Figura 4 – Representação simplificada de uma AST

Outra estrutura comum na engenharia de compiladores é a Tabela de Símbolos (*Symbols Table*) (AHO et al., 2007). Ela é construída durante a fase de análise léxica e contém todos os símbolos declarados em cada escopo do código-fonte. Exemplos, todas as variáveis declaradas em uma função, ou os atributos de uma classe e de cada um de seus métodos. Normalmente a tabela contém, além dos identificadores, o tipo de dados de cada símbolo.

2.4 Trabalhos Relacionados

Rodriguez-Echeverria et al. (2018) propõe a implementação de uma infraestrutura LSP com foco em modelagem gráfica. Seu objetivo é desacoplar o suporte à edição da linguagem gráfica em si. Para isso, estes autores estudaram diferentes alternativas para a utilização de clientes LSP e servidores. Diferentemente, nosso trabalho foca nas preocupações gerais e práticas para implementar servidores LSP, inclusive estudando somente linguagens baseadas em *parsing*.

O trabalho de Erdweg et al. (2013) apresenta detalhes de *language workbenches* para o desenvolvimento de GPLs e DSLs. Eles investigam a implementação de 10 *language*

workbenches com respeito a cobertura de funcionalidades, tamanho, e dependências necessárias. Porém, nada é explorado em relação ao suporte à edição. Adicionalmente, esse trabalho está ultrapassado (de 2013) em comparação aos recentes avanços na engenharia de linguagem e suporte à edição, relembrando que o LSP surgiu somente em 2016.

Bünder e Kuchen (2019) apresenta um caso de uso onde seis funcionalidades chave do LSP foram implementadas para uma DSL criada com o *language worbench* XText. O XText oferece suporte nativo ao LSP e um *generator* que cria automaticamente um *Language Server* em Java, baseado no LSP4J (uma SDK para implementação de servidores LSP). Para consumir o servidor foram também implementados dois clientes, um para a IDE Theia e outro para a IDE Eclipse. Neste trabalho foi avaliado o esforço necessário para implementar esses clientes, avaliado como moderado pelos pesquisadores. No contexto do LSP Bünder e Kuchen (2020) apresenta como este protocolo pode ser utilizado para satisfazer ambos, os desenvolvedores e especialistas de domínio, com diferentes preferências, podendo trabalhar no mesmos projetos. O LSP fornece formas de integrar diferentes editores no desenvolvimento de projetos dirigidos por modelos.

Em um livro recente Gunasinghe e Marcus (2022), desenvolvedores do servidor que suporta a linguagem Ballerina, descrevem os requisitos para implementar o suporte à edição utilizando o LSP. Para tal, eles detalham a implementação de ambos cliente e servidor. Ambos os trabalhos (BÜNDER; KUCHEN, 2019; BÜNDER; KUCHEN, 2020; GUNASINGHE; MARCUS, 2022) são relatórios de experiência para um cenário específico ou tecnologia, com objetivos distintos dos deste trabalho, de explorar aspectos de implementação considerando múltiplos servidores.

Visando aproveitar informações de tempo de execução, Stolpe et al. (2019) propõe a utilização do Framework Truffle, utilizado pela GraalVM para suportar linguagens dinâmicas. A abordagem aproveita a API do Truffle para gerar, interpretar e manipular ASTs com foco em linguagens dinâmicas. Coletando informações de tempo de execução para inferir detalhes como os tipos dos símbolos. A técnica proposta visa fornecer um suporte à edição independente da linguagem em casos onde não haja um servidor de linguagem especifico, garantindo assim a sua atratividade.

Mészáros, Cserép e Fekete (2019) analisam quão frequente determinadas funcionalidades de compreensão de código (ex., `Goto Definition`) são encontradas em IDEs amplamente utilizadas no mercado. O mesmo trabalho demonstra um caso onde uma ferramenta de `Code Comprehension` chamada CodeCompass é estendida para servir como implementação de uma funcionalidade do LSP. Essa ferramenta gera um diagrama UML que a IDE pode apresentar ao desenvolvedor, o auxiliando na compreesão do código-fonte.

Com propósito similar, Pupo et al. (2019) trata de uma ferramenta que utiliza Aprendizado de Máquina para aprimorar a segurança de aplicações Javascript. As funcionalidades da ferramenta são disponibilizadas para a IDE através das funcionalidades

`Diagnostics` e `Execute Command` do LSP. Através dessas funcionalidades o cliente pode solicitar que um comando não especificado pelo protocolo seja executado em uma determinada posição do código fonte. O comando consiste em escanear o código para encontrar pontos de interesse que devem ter a segurança aprimorada. Dois aspectos interessantes a serem citados do trabalho são: (i) referências a quais funcionalidades do LSP foram utilizadas, e (ii) a necessidade de utilizar bibliotecas externas para gerar a AST do código-fonte, essa AST é atravessada em busca de nós do tipo *callExpression*.

Uma proposta de solução é apresentada por Coulon et al. (2020) que combina uma abordagem generativa com *Feature Models* para criar e configurar uma IDE baseada em microsserviços que implementam as funcionalidades de um Servidor de Linguagem. Após a implementação, a performance da IDE foi validada. Foi constatada uma melhor performance para serviços como *rename* e *compile*, em comparação a uma implementação monolítica da IDE. Os autores atribuem a melhor performance à natureza distribuída do projeto. Dada a necessidade de carregar os modelos a cada requisição, observa-se também a necessidade de implementação de otimizações de performance, como a incorporação de cache a arquitetura.

Esses trabalhos focam principalmente em descrever a implementação para casos de uso específicos, sem discutir preocupações ou práticas genéricas, ou mesmo padrões, que devem ser levadas em consideração pelos desenvolvedores ao arquitetar/implementar seus projetos.

3 Metodologia

Este estudo foca na identificação de aspectos de implementação do suporte à edição em servidores LSP existentes. Considerando como aspectos, os *interesses relevantes que devem ser considerados quando os desenvolvedores implementam/arquitetam o suporte à edição para as linguagens que eles suportarão*. Foram identificadas também as práticas para lidar com esses aspectos. A Figura 5 ilustra a metodologia adotada. No lado esquerdo da figura, a *coleta de dados*, onde a amostra utilizada para a pesquisa foi selecionada, descrita na Seção 3.1. A *análise quantitativa*, ilustrada no centro da figura e descrita na Seção 3.2. Por fim, a *análise qualitativa*, ilustrada no lado direito da figura e descrita na Seção 3.3. Detalhes adicionais foram disponibilizados em nosso pacote de replicação online (BARROS et al., 2022).

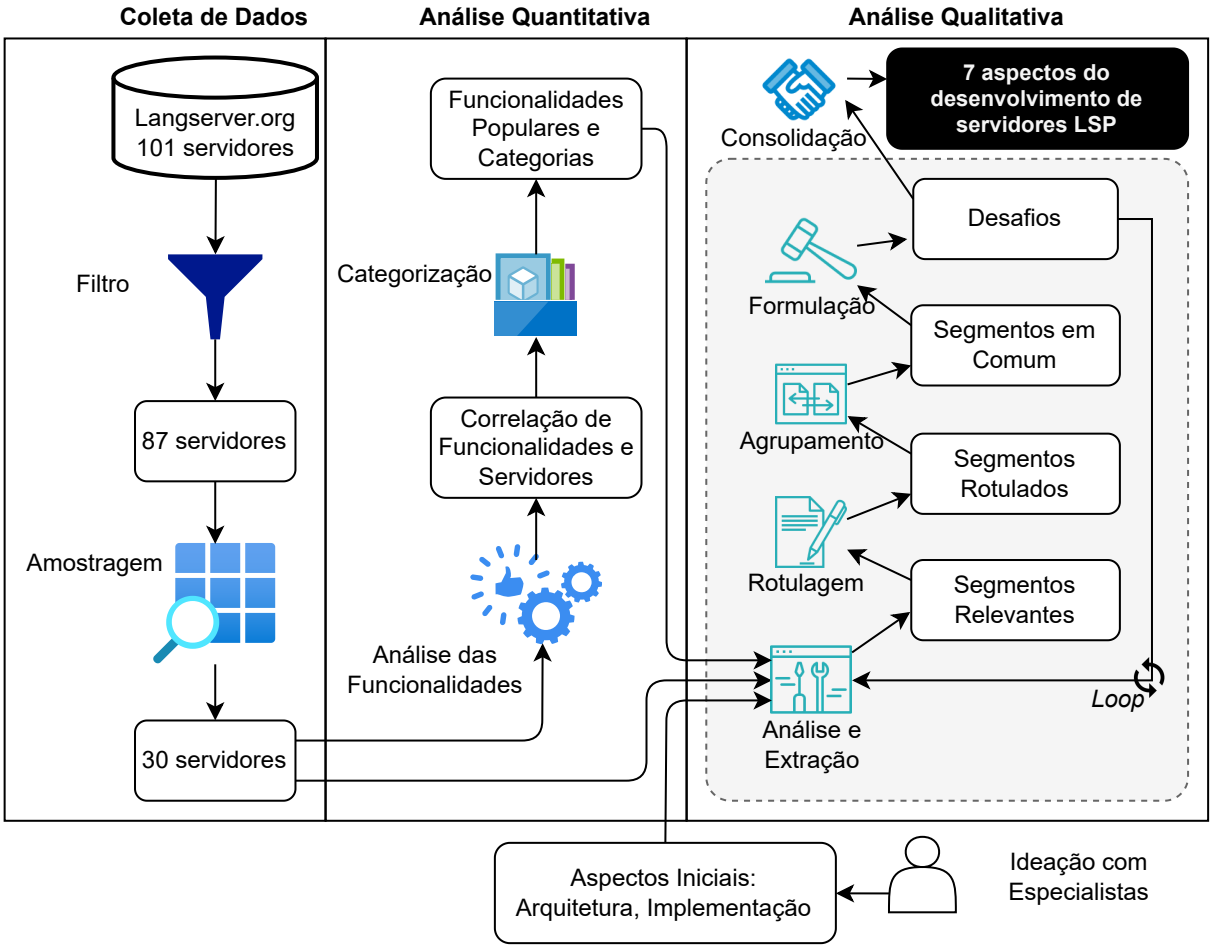

Figura 5 – Visão geral da metodologia

3.1 Coleta de dados

Como objetos de estudo para o trabalho trabalho, foram selecionados 30 servidores LSP. Sendo esses, uma amostra relevante do conjunto de servidores contidos em um repositório curado e mantido pela comunidade, que cataloga servidores LSP conhecidos (SOURCEGRAPH, 2022). Atualmente o repositório contém 125 LSPs (Dezembro 2022), porém, no momento da seleção da amostra continha 101 LSPs (Março 2021). A amostra foi filtrada/selecionada de acordo com os seguintes critérios:

Código-fonte inacessível: Primeiro, foram excluídos todos os servidores LSPs para o qual seus códigos fonte não foram encontrados. Afetando 3 LSPs.

Não mantidos ativamente: Também foram excluídos 11 LSPs listados como arquivados ou depreciados. Essa informação foi coletada do repositório da comunidade, bem como dos repositórios de código-fonte dos servidores avaliados (ex., arquivos README ou rotulados como depreciados no Github). Esse passo garantiu o foco no estado da prática do desenvolvimento de servidores LSP.

Servidores LSP implementados com a linguagem Java: Foram selecionados todos os servidores LSP implementados em Java (12 servidores) por quatro razões: (i) é a linguagem de maior domínio do autor e dos colaboradores do trabalho, facilitando a compreensão das implementações avaliadas; (ii) existe um ferramental considerável de engenharia de linguagem que é implementado em Java, permitindo a comparação dos LSPs com a experiência dos colaboradores do trabalho na criação de DSLs utilizando ferramentas baseadas em Java. Especificamente, EMF, o gerador de *parsers* ANTLR e outros *frameworks* relacionados para análise de linguagem e transformação de modelos ou programas (BERGER et al., 2016; DRAGULE et al., 2021; HEBIG et al., 2018; LILLACK; BERGER; HEBIG, 2016; PELDSZUS, 2022; PELDSZUS et al., 2015; QUEIROZ; BERGER; CZARNECKI, 2019; SCHAUSS et al., 2017; STRÜBER; PELDSZUS; JÜRJENS, 2018; VÖLTER et al., 2016; WASOWSKI; BERGER, 2022); (iii) Java está entre as linguagens de programação mais populares e bem compreendidas, facilitando a compreensão de seus códigos, em oposição a análise de LSPs implementadas em R, Go, Erlang ou Elixir, por exemplo; (iv) o Java em conjunto com o Typescript é a linguagem de programação mais popular entre todos os servidores documentados (SOURCEGRAPH, 2022).

Seleção aleatória entre os servidores LSP restantes: Adicionalmente ao entendimento dos detalhes necessários para a implementação de servidores baseado nas implementações em Java, a análise continuou com uma amostra mais diversa. Através da seleção aleatória de mais servidores, até alcançar um tamanho de amostra total de 30 servidores, incluindo então 18 servidores adicionais.

O propósito desse processo de amostragem foi controlar a seleção dos servidores, com o intuito de habilitar uma exploração criteriosa entre os servidores existentes. A

Tabela 2 provê uma visão geral dos servidores LSP estudados, listando os 30 servidores (≈30% da lista curada). Os servidores da amostra são implementados em 14 linguagens e suportam um total de 31 linguagens de programação diferentes. Levando em consideração a saturação (GLASER, 1978) como uma critério de qualidade, a mesma foi alcançada depois da análise do 18º servidor da nossa amostra, não encontrando nenhum novo tópico ou código após esse servidor (explicado na Seção 3.3).

3.2 Análise Quantitativa

Essa etapa consistiu em primeiramente, obter uma visão geral das funcionalidades LSP implementadas e da popularidade dessas funcionalidades entre os servidores da amostra. Um segundo objetivo foi avaliar as correlações entre essas funcionalidades, baseado tanto na identificação de funcionalidades implementadas em conjunto, quanto em sua popularidade de acordo com caraterísticas das linguagem suportada (ex., paradigma). Para identificar a popularidade de cada funcionalidade contamos quantas vezes elas foram implementadas pelos servidores da amostra. O motivo para a identificação das funcionalidades populares foi prover um subconjunto homogêneo de implementações para posterior análise qualitativa. Foram também identificadas possíveis categorias de suporte à edição, basado no propósito dessas funcionalidades. Essa análise é ilustrada no centro da Figura 5.

Para contar a quantidade de vezes que uma funcionalidade é implementada, utilizamos uma informação estabelecida pela especificação LSP e disponível por padrão em cada servidor. Quando o cliente inicia a comunicação com o servidor, ele requisita um método chamado *initialize* (MICROSOFT, 2022b). O método em questão retorna um objeto implementando a interface *InitializeResult* (Figura 6). O corpo da resposta descreve através da propriedade *capabilities* todas as funcionalidades disponibilizadas pelo servidor. Logo, para cada servidor, o código-fonte que implementa o método *initialize* foi inspecionado para então coletar a lista de funcionalidades implementadas. Esse mesmo método retorna também a versão da especificação LSP utilizada por cada servidor. No momento em que essa análise foi realizada, a versão da especificação LSP corrente era a *3.16.0*, lançada em 12/2020. Sendo essa uma versão recente e vários servidores da amostra não a tendo implementado, foram consideradas funcionalidades definidas somente até a versão *3.15.0*, de Janeiro/2020.

3.3 Análise Qualitativa

Foram identificados durante essa etapa os diferentes aspectos e práticas para o desenvolvimento de servidores LSP. Focando inicialmente em como estruturar um servidor LSP a nível arquitetural bem como em aspectos comuns a nível de implementação. Sendo esses os dois aspectos inicias da Figura 5.

Tabela 2 – Relação de servidores LSP analisados

	Linguagem	Tipo	Implem.	Repositório
1	ActionScript3	GPL	Java	<github.com/BowlerHatLLC/vscode-nextgenas>
2	R	GPL	R	<github.com/REditorSupport/languageserver>
3	Go	GPL	Go	<github.com/golang/tools>
4	Erlang	GPL	Erlang	<github.com/erlang-ls/erlang_ls>
5	Flux	DSL	Rust	<github.com/influxdata/flux-lsp>
6	Assembler	GPL	C++	<github.com/eclipse/che-che4z-lsp-for-hlasm>
7	XML	DSL	Java	<github.com/angelozerr/lsp4xml>
8	Turtle	DSL	Type-script	<github.com/stardog-union/stardog-language-servers>
9	C/C++	GPL	C++	<github.com/MaskRay/ccls>
10	Java	GPL	Java	<github.com/eclipse/eclipse.jdt.ls>
11	Robot Framework	DSL	Python	<github.com/robocorp/robotframework-lsp>
12	Rust	GPL	Rust	<github.com/rust-analyzer/rust-analyzer>
13	Xtext (any lang.)	DSL	Java	<github.com/eclipse/xtext-core>
14	Python	GPL	Python	<github.com/palantir/python-language-server>
15	Cobol	GPL	Java	<github.com/eclipse/che-che4z-lsp-for-cobol>
16	Elixir	GPL	Elixir	<github.com/elixir-lsp/elixir-ls>
17	Puppet	DSL	Ruby	<github.com/lingua-pupuli/puppet-editor-services>
18	PHP	GPL	PHP	<github.com/felixfbecker/php-language-server>
19	Groovy	GPL	Java	<github.com/prominic/groovy-language-server>
20	LaTeX	DSL	Rust	<github.com/efoerster/texlab>
21	Apache Camel	DSL	Java	<github.com/camel-tooling/camel-language-server>
22	Ballerina	GPL	Java	<github.com/ballerina-platform/ballerina-lang>
23	Java	GPL	Java	<github.com/georgewfraser/vscode-javac>
24	SonarLint	DSL	Java	<github.com/SonarSource/sonarlint-language-server>
25	Lua	GPL	Java	<github.com/EmmyLua/EmmyLua-LanguageServer>
26	MOCA	DSL	Java	<github.com/mrglassdanny/moca-language-server>
27	OCaml	GPL	OCaml	<github.com/ocaml/ocaml-lsp>
28	TTCN-3	DSL	Go	<github.com/nokia/ntt>
29	Swift & C-family	GPL	Swift	<github.com/apple/sourcekit-lsp>
30	Vala	GPL	Vala	<gitlab.gnome.org/esodan/gvls>

3.3.1 Análise Temática Exploratória

Uma análise temática foi aplicada como método de trabalho (CRUZES; DYBA, 2011). Durante a análise, o conjunto de aspectos iniciais foi refinado e expandido, assim

```
return InitializeResult(capabilities: ServerCapabilities(
  textDocumentSync: TextDocumentSyncOptions(
    openClose: true,
    change: .incremental,
    willSave: true,
    willSaveWaitUntil: false,
    save: .value(TextDocumentSyncOptions.SaveOptions(includeText: false))),
  hoverProvider: true,
  completionProvider: CompletionOptions(
    resolveProvider: false,
    triggerCharacters: [".", "("]),
  definitionProvider: nil,
  implementationProvider: .bool(true),
  referencesProvider: nil,
  documentHighlightProvider: true,
  documentSymbolProvider: true,
  codeActionProvider: .value(CodeActionServerCapabilities(
    clientCapabilities: initialize.capabilities.textDocument?.codeAction,
    codeActionOptions: CodeActionOptions(codeActionKinds: [.quickFix, .refactor]),
    supportsCodeActions: true)),
  colorProvider: .bool(true),
  foldingRangeProvider: .bool(true),
```

Figura 6 – Exemplo parcial do *InitializeResult* para o servidor Swift (#29)

como as práticas de implementação dos LSPs. Essa análise é ilustrada no lado direito da Figura 5.

Durante essa etapa, uma análise exploratória (EASTERBROOK et al., 2008) foi conduzida. O primeiro passo para identificar as práticas de implementação utilizadas frequentemente foi coletar e catalogar iterativamente segmentos de código-fonte relevantes em servidores existentes, focando tanto na implementação das funcionalidades de edição, quanto na estruturação do servidor LSP a nível arquitetural. Após, foram identificadas e agrupadas de forma sistemática as práticas de implementação comuns, baseadas nos segmentos identificadas, avaliando todos os servidores da amostra. O agrupamento de práticas comuns forma o conjunto aspectos de apresentados no trabalho.

Para todos os servidores LSP, foi analisada a estrutura geral do projeto e então identificados os pontos de entrada de cada uma das funcionalidades selecionadas para inspeção. Um exemplo de ponto de entrada é a implementação do método *completion* na classe *CompletionProvider* do servidor ActionScript3 (#1) (Tabela 2). Uma vez com esses pontos de entrada identificados, se iniciou uma análise extensiva do código-fonte para identificar as práticas de implementação. A avaliação do código seguiu um processo inspirado pelos passos da análise temática (CRUZES; DYBA, 2011). Para ser mais preciso, como demonstrado na Figura 5, a análise partiu de dois aspectos iniciais genéricos (Arquitetura e Implementação), e então continuou como descrito nas seções seguintes:

3.3.2 Extração de Segmentos Relevantes

Primeiro, foram analisadas cada uma das implementações de servidores LSP da amostra e extraídos *Segmentos Relevantes*. O código-fonte de cada funcionalidade selecionada foi avaliado completamente para a extração de porções relevantes para os dois aspectos iniciais (arquitetura e implementação). Além de analisar o método de entrada para a implementação da funcionalidade, foram também seguidas as chamadas de métodos para outras classes/arquivos do código-fonte dependentes, também foram avaliadas as documentações de bibliotecas externas quando o propósito do código chamado não podia ser identificado explicitamente pela leitura inicial.

```
30  pgf, err := snapshot.ParseGo(ctx, fh, ParseFull)
31  if err != nil {
32      return nil, err
33  }
```

Figura 7 – Exemplo de segmento relevante do código-fonte extraído da implementação da funcionalidade `Format` no servidor Go (#3)

A Figura 7 demonstra um exemplo de segmento de código relevante onde ocorre uma chamada para o método *ParseGo* da estrutura *snapshot*. Essa implementação executa a ferramenta *Go/buildTools* para gerar uma representação intermediária do código-fonte contido no arquivo referenciado pela variável *fh* (linha 30).

Segmentos de código relevantes são aqueles que demonstram detalhes sobre como o servidor é arquitetado ou implementado, porém, focando-se em aspectos que não fossem específicos das funcionalidades. Exemplos de aspectos relevantes são todos os códigos próprios ou de terceiros que auxiliam em atividades como:

- Implementação do protocolo;
- Representação de instâncias que serão manipuladas (ex., arquivos, códigos-fonte, e símbolos);
- Busca de informações sobre os símbolos contidos no código-fonte;
- Coleta de dados sobre o ecossistema de desenvolvimento (ex., variáveis de ambiente, bibliotecas utilizadas no projeto);
- Transformação de dados e processamento das funcionalidades através das manipulação das instâncias.

Os segmentos relevantes foram então traduzidos em uma afirmação em linguagem natural descrevendo qual aspecto de implementação/arquitetura será destacado. No

exemplo de código da Figura 7, uma descrição possível seria: *it parses the file using the "Go/buildTools."*. Cada segmento é descrito com pelo menos uma sentença.

Rotulagem dos Segmentos Extraídos: Cada segmento relevante foi então *rotulado*, baseado em qual aspecto arquitetural ou prática de implementação ele descreve. Esses rótulos são os códigos da análise temática. No exemplo apresentado anteriormente, o rótulo foi **COMPILER_PARSER**, que descreve a utilização de um compilador externo para a tradução da instância código-fonte para uma representação intermediária. O conjunto de códigos/rótulos identificados é descrito na Tabela 3. A mesma sentença, porém com diferentes sentenças, pode ter mais de um rótulo.

Tabela 3 – Relação de códigos identificados

Código (Grupo)	**Descrição**
LAYERED_ARCHITECTURE	Descreve quando alguma estratégia de separação em camadas é utilizada para organizar a base de código e separar as responsabilidades de implementação
COARSE (GRANULARITY)	Descreve detalhes de nível mais alto sobre as implementações das funcionalidades. Por exemplo, se a execução é delegada para bibliotecas externas ou se são utilizados arquivos/classes por funcionalidade
FINE (GRANULARITY)	Descreve detalhes mais refinados sobre o conteúdo dos arquivos/classes que implementam as funcionalidades
COMPILER_PARSER (LIBRARY)	Descreve se o servidor utiliza alguma biblioteca do tipo Compilador/Parser
EMBEDDED_GRAMMAR (LIBRARY)	Descreve se o projeto do servidor contêm alguma gramática embarcada
DOCUMENTATION_HANDLER (LIBRARY)	Descreve se o servidor utiliza qualquer biblioteca para coletar documentações sobre o código-fonte (ex., JavaDoc).
ENVIRONMENT_INFO (LIBRARY)	Descreve se o servidor utiliza qualquer biblioteca para coletar informações sobre o ambiente onde o projeto se aplica (ex., versão da linguagem, informações de tempo de execução, gestão de dependências).

LINTER (LIBRARY)	Descreve se o servidor utiliza alguma ferramenta de *Linting* ou de Análise Estática de Código para coletar diagnósticos da sintaxe do programa/modelo
LSP_SDK (LIBRARY)	Descreve se o servidor utiliza alguma SDK para auxiliar na implementação do LSP
LANGUAGE_SUPPORTING (LIBRARY)	Descreve se o servidor delega completamente a execução das funcionalidades para qualquer biblioteca que as implemente parcial ou completamente
AST (INSTANCE)	Descreve se o servidor utiliza qualquer representação abstrata (ex., Árvores Abstratas de Sintaxe, XML DOM) ou equivalente para representar as instâncias do programa/modelo
TREE_GRAPH (INSTANCE:TRAVERSAL)	Descreve quando o servidor utiliza qualquer algoritmo de Grafo (ex., Busca em Largura, Busca em Profundidade) ou de Árvore (ex., *in-order, pre-order, post-order*) para a travessia da representação abstrata do programa/-modelo
VISITOR (INSTANCE:TRAVERSAL)	Descreve quando o servidor utiliza o padrão *Visitor* para a travessia da representação abstrata do programa/modelo
AD_HOC (INSTANCE:TRAVERSAL)	Descreve quando o servidor utiliza algoritmos simples (*adhoc*) para a travessia das representação abstrata do programa/modelo (ex., *bottom-up*, navegação entre os nós irmãos).
THIRD_PARTY (INSTANCE:TRAVERSAL)	Descreve quando o servidor atravessa a representação abstrata do programa/modelo com a ajuda de bibliotecas de terceiro
AUXILIAR_INDEX_STRUCTURE (OPTIMIZATION)	Descreve quando o servidor utiliza estruturas auxiliares para indexar os nós relevantes da representação abstrata (ex., Tabelas de Símbolos).

CONTEXT (ENCLOSING_SCOPE_SCANNING)	Descreve quando o servidor utiliza algoritmos de manipulação de texto para escanear o código-fonte diretamente e então identificar o contexto onde o cursor se aplica (ex., se o cursor está dentro de um bloco de comentários ou entre parenteses).
TARGET_TOKEN (ENCLOSING_SCOPE_SCANNING)	Descreve quando o servidor utiliza algoritmos de manipulação de texto para escanear o código-fonte diretamente e verificar qual o símbolo que está na mesma posição do cursor
THIRD_PARTY (ENCLOSING_SCOPE_SCANNING)	Descreve quando o servidor utiliza recursos disponibilizados por bibliotecas de terceiro (ex., *Parser/Compiler*) para escanear o código-fonte diretamente e identificar tanto contextos quanto símbolos.
CHANGE_TRACKING	Descreve quais ações o servidor executa quando mudanças acontecem com o programa/modelo. Tanto na execução da função *"didChange"* quanto na *"didSave"*, estabelecidas pelo protocolo

Agrupamento dos Segmentos Comuns: Além do agrupamento dos segmentos comuns por rótulos, práticas relevantes são descritas em um nível mais alto como *temas de codificação*. Esses temas são os grupos da Tabela 3 e no processo teórico do trabalho, são desafios que podem ser implementados utilizando diferentes estratégias. Nessa relação, diferentes rótulos podem pertencer ao mesmo grupo (ex. diferentes rótulos para o grupo *Library*, na tabela.

Formulação dos Desafios e Estratégias de Implementação: A combinação de grupos e códigos consolidados foi traduzida para a teoria do trabalho, que descreve os aspectos (grupos) e possíveis formas de implementa-los (rótulos/códigos). Onde cada aspecto é um desafio relevante para arquitetar ou implementar o suporte à edição para uma linguagem de software. Resultados apresentados no Capítulo 4.

Processo Iterativo: A formulação de códigos e agrupamento ocorreu iterativamente para cada servidor e funcionalidade. Após finalizado um servidor esse processo seguiu com o próximo selecionado, até que finalizada a análise toda a amostra.

4 Resultados

Os desafios para a implementação do suporte à edição nos servidores LSP foram identificados através da análise de aspectos típicos de implementação e arquitetura de software. Foram considerados como aspectos, a forma com que o sistema é estruturado, seguindo então de práticas de implementação do sistema que atendam a esses aspectos. Focando na engenharia de linguagem, um exemplo de prática relevante observada é a forma como as instâncias de codificação são representadas em um servidor (ex., utilização de ASTs para representar de forma abstrata a estrutura do código fonte, ou a utilização de uma simples representação como texto puro).

A seguir são apresentados (i) uma visão geral de como em sua maioria as funcionalidades de suporte à edição são implementadas; (ii) os dois aspectos baseados na análise quantitativa; e (iii) os cinco aspectos finais que emergiram e foram respondidos durante a análise qualitativa.

4.1 Fluxo de trabalho do LSP

Ao observar as implementações das funcionalidades, nota-se que cada requisição de funcionalidade é independente das outras e consiste em quatro partes que são descritas na (Figura 8):

- **1. Requisição da funcionalidade:** Quando requisitando uma funcionalidade, o Cliente informa a ação que deve ser executada (ex., *textDocument/completion*) e o documento alvo que será operado. Para descrever esse alvo o cliente envia o identificador do documento (*DocumentID*) e a posição alvo (ex., linha e carácter do cursor, ou intervalo de linhas e colunas) no documento. O servidor então, com essas informações, consegue identificar qual o código que deverá ser executado (ex., classe ou método) para atender a requisição, e resolver os parâmetros necessários para executar o mesmo.
- **2. Carregamento do arquivo:** Baseado no *DocumentID*, o servidor deve carregar o conteúdo do arquivo alvo (ex., código-fonte). Logo, não somente o cliente onde o arquivo está sendo editado como também o servidor necessitam de acesso ao arquivo.
- **3. Execução da ação:** Tendo as informações da requisição e acesso ao arquivo do código-fonte, o servidor então executa a funcionalidade (ex., identificar qual símbolo deve ser completado e coletar as sugestões de código). Algumas funcionalidades, como a `Completion` ou `Rename`, requerem que o servidor explore todos os documentos

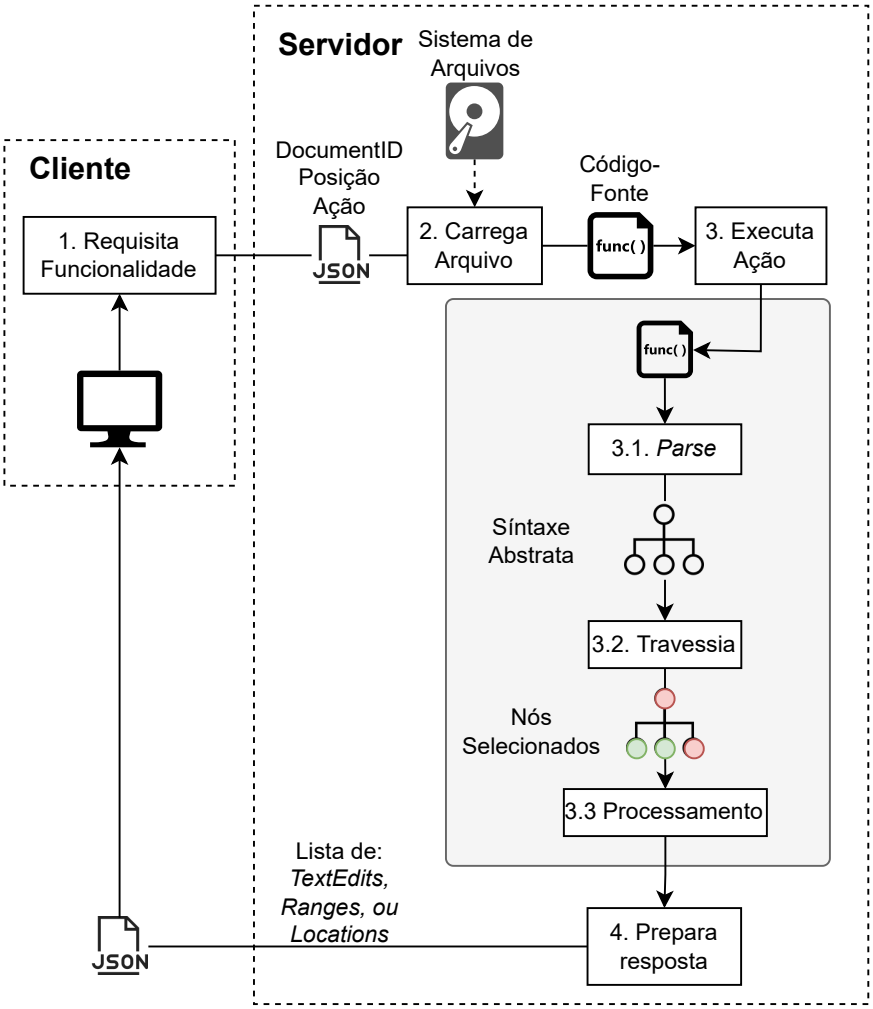

Figura 8 – Visão geral do fluxo de processamento de uma requisição

do espaço de trabalho bem como as bibliotecas externas utilizadas. Essa etapa foi separada em três passos complementares:

  - **3.1 *Parsing:*** primeiro o arquivo é interpretado e transformado em uma representação intermediária.
  - **3.2 Travessia:** então a representação resultante pode ser explorada para coletar as informações necessárias.
  - **3.3 Processamento:** ao final um processamento adicional das informações coletadas pode ser realizado, para filtrar as informações que efetivamente serão enviadas para o cliente.

- **4. Preparação da resposta:** Finalmente o servidor retorna a mensagem de resposta. Por exemplo, para a funcionalidade `Completion`, depois de coletar as sugestões que podem ser inseridas na posição desejada, o servidor responde com o conjunto de itens possíveis. Cada item é do tipo *TextEdit*, representando uma operação de edição à ser aplicada caso determinado item seja selecionado.

Apesar de ser um fluxo comum, não necessariamente todas essas etapas são necessárias para cada funcionalidade. Nuances de implementação serão apresentadas nos próximos desafios como: (i) utilização de cache, para evitar o carregamento e *parsing* do código-fonte a cada requisição, ou (ii) a utilização de bibliotecas externas para a implementação das funcionalidades (sem a necessidade de executar os três passos relacionados), dentre outras.

4.2 Aspecto 1: Seleção das funcionalidades de edição

Ao implementar um novo servidor LSP, é essencial saber quais funcionalidades são disponibilizadas na prática, pois a implementação de outras com pouca aderência pode ser considerado um desperdício de esforço e recursos. Para obter uma visão geral das funcionalidades disponibilizadas pelos servidores existentes, a amostra foi avaliada. Após coletar as funcionalidades, elas foram analisadas para entender as semelhanças e diferenças entre diferentes suportes a edição. Na versão da especificação utilizada no momento da análise, eram 23 as funcionalidades descritas pelo protocolo (excluindo funcionalidades que somente incrementavam outras através de um passo adicional, por exemplo, como a requisição de informações adicionais sobre os itens retornados ao executar a funcionalidade `Goto Definition`).

4.2.1 Funcionalidades implementadas com maior frequência

A Figura 9 demonstra quais funcionalidades são implementadas por cada servidor. As funcionalidades estão listadas nas colunas e os servidores nas linhas. Ambos ordenados de forma ascendente pelo número de funcionalidades implementadas ou de servidores as implementando, respectivamente (totalizadores na última linha/coluna). Entender a popularidade das funcionalidades de edição individualmente bem como os *outliers* pode auxiliar os engenheiros na decisão de quais funcionalidades implementar primeiro, bem como os pesquisadores a focarem seus trabalhos nas necessidades da industria. As funcionalidades providas foram avaliadas de acordo com a categoria de suporte à edição e popularidade entre os servidores da amostra.

`Goto Definition` e `Diagnostics` são as funcionalidades implementadas com maior frequência. Essas funcionalidades constituem um suporte à edição comum para linguagens (GUNASINGHE; MARCUS, 2022). Ainda sobre a funcionalidade `Diagnostics`, sua popularidade pode ser explicada, em parte, pela facilidade de implementá-la, pois os Compiladores/*Parsers* habitualmente fornecem o resultado do diagnóstico em conjunto com a representação gerada pelo processo de *parsing* (mais informações na Seção 4.3).

Somente dois servidores não implementam essas funcionalidades, os servidores #24 e #8 para a funcionalidade `Goto Definition` e os servidores #24 e #28 para a funcionalidade `Diagnostics`. Esses servidores utilizam o LSP de uma maneira própria, muitas vezes

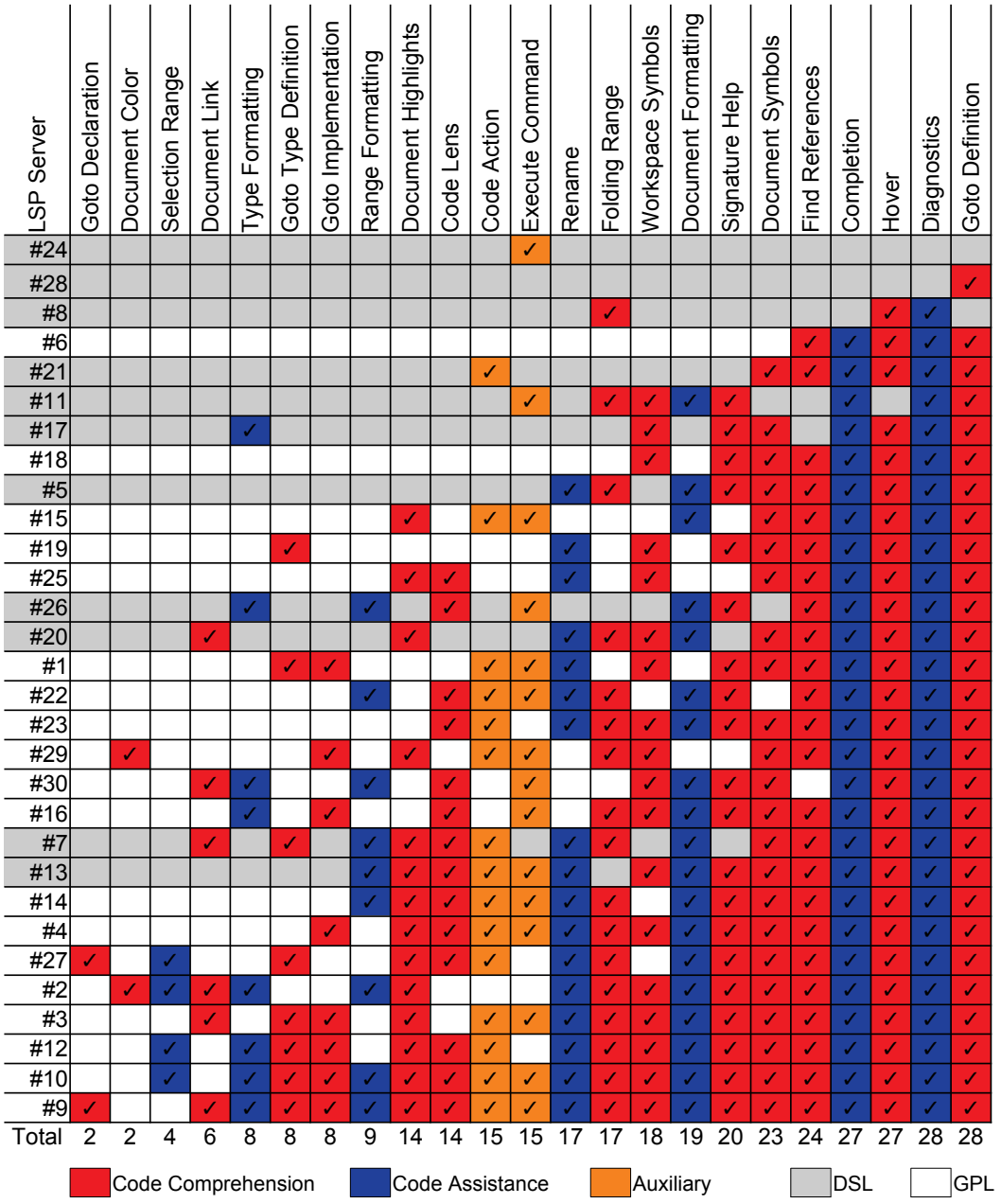

Figura 9 – Funcionalidades implementadas em cada servidor LSP

sem a necessidade de prover muitas funcionalidades. Um exemplo de como os servidores podem ser implementados, e não necessitarem implementar diversas funcionalidades, é quando esses servidores somente provêm serviços adicionais, como refatorações ou análise estática de código. O servidor #24 demonstra essa necessidade, onde o mesmo tem como único propósito disponibilizar ao editor a integração com a ferramenta *Sonar* [1].

A seguir, porém ainda na análise quantitativa, são apresentadas as funcionalidades mais populares, considerando como populares as que são implementadas por pelo menos dois terços dos servidores (20 servidores), totalizando então 7 funcionalidades. Reduzir o número de funcionalidades tem como intuito diminuir o escopo de análise e então permitir a

[1] <https://www.sonarsource.com/products/sonarlint/>

inspeção de suas implementações em profundidade. O segundo motivo para a identificação das funcionalidades populares é garantir que o conjunto a ser analisado será homogêneo, evitando assim que funcionalidades com pouca aderência, porém implementações peculiares, produzam um viés nos resultados. A seguir, são sumarizadas as sete funcionalidades mais populares, bem como, suas categorias de suporte à edição e a quantidade de servidores que as implementam.

Goto Definition (*Code Comprehension*, 28 servidores): permite que os usuários naveguem pelo projeto para encontrar a posição onde a referenciada instância do elemento foi definida. Recebe como parâmetro o identificador do documento aberto e a posição (linha e coluna) do cursor (Figura 10).

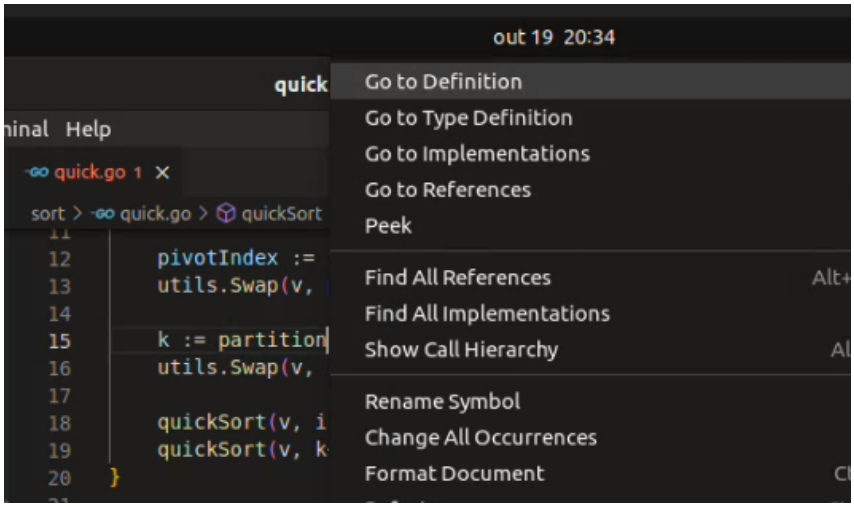

Figura 10 – Exemplo de utilização da funcionalidade `Goto Definition` - Disparo

O usuário solicita a localização onde a função *partition*, chamada na linha 15 e coluna 14 do arquivo quick.go, foi definida.

É retornado pelo servidor a posição onde a instância foi definida (ex., onde a variável contida no cursor foi declarada), contendo (i) o identificador do documento onde a instância foi definida; e (ii) o intervalo da declaração (linhas e colunas iniciais e finais) (Figura 11).

Diagnostics (*Code Assistance*, 28 servidores): retorna avisos/erros de compilação, *parsing* ou mesmo de ferramentas de inspeção de códigos. A funcionalidade de diagnósticos não é requisitada diretamente pelo cliente, porém retornada automaticamente pelo servidor ao compilar ou alterar qualquer arquivo aberto. A notificação com o resultado dos diagnósticos é enviada para o cliente contendo (i) o identificador do documento; e (ii) uma lista de diagnósticos. Cada um dos diagnósticos retornados contém o intervalo do código com erro e uma mensagem descritiva (Figura 12).

Hover (*Code Comprehension*, 27 servidores): exibe informações relacionadas a posição requisitada no documento de texto, geralmente ao posicionar o mouse sobre um

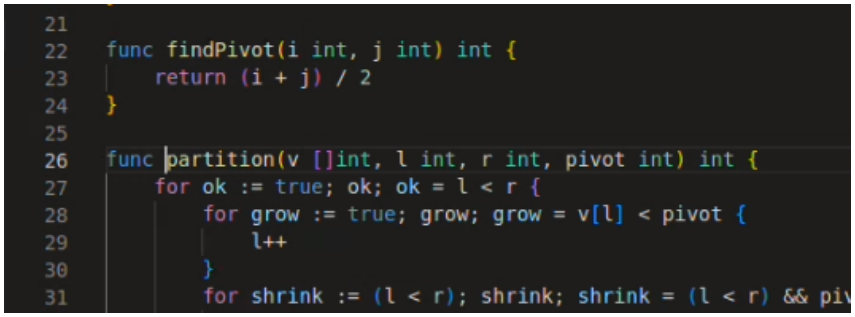

Figura 11 – Exemplo de utilização da funcionalidade `Goto Definition` - Resultado

O servidor então retorna que a função *partition* foi declarada no arquivo (quick.go), porém na linha 26 e intervalo 5 a 14 (colunas). Com essa informação em mãos o cliente consegue navegar para o local indicado.

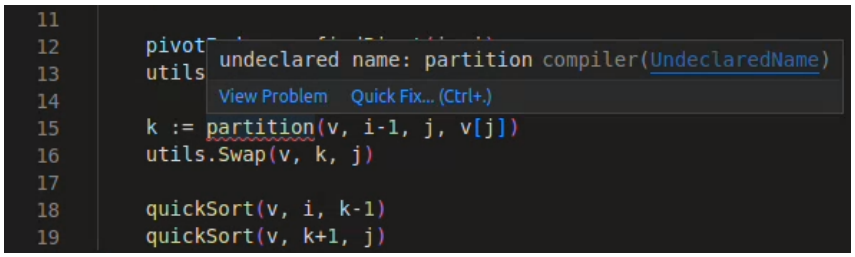

Figura 12 – Exemplo de utilização da funcionalidade `Diagnostics`.

Após alterar o arquivo o servidor executa a análise de diagnósticos e retorna que na linha 15 do arquivo quick.go e intervalos de colunas 7 a 16 há um erro de compilação, onde a função *partition* não foi declarada.

símbolo. As informações retornadas são definidas pelo servidor e dependem do tipo do símbolo. Por exemplo, em caso do símbolo alvo ser uma variável, pode retornar o seu tipo, no caso de uma função, pode retornar sua assinatura.

A funcionalidade `Hover` recebe como parâmetro (i) o identificador do documento; e (ii) a posição do cursor/mouse (linha e coluna). O retorno contém texto puro ou *markup* com os detalhes do símbolo (Figura 13).

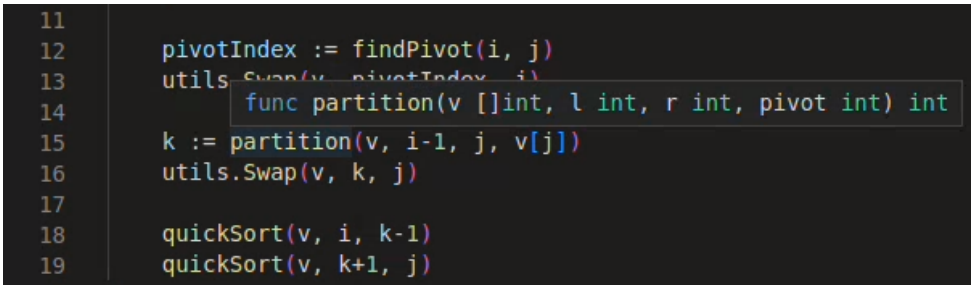

Figura 13 – Exemplo de utilização da funcionalidade `Hover`.

O usuário posiciona o cursor sobre o símbolo *partition* (linha 15 e coluna 14 do arquivo quick.go). O cliente então requisita ao servidor detalhes sobre o símbolo nessa posição. O servidor retorna a assinatura da função *partition*.

Completion (*Code Assistance*, 27 servidores): computa uma lista de itens sugeridos

para completar uma posição do cursor, onde os itens são apresentados na interface do usuário. O cliente informa a posição do cursor onde espera as sugestões de símbolos, contendo (i) a identificação do arquivo; e (ii) a posição do cursor (linha e coluna). O servidor então retorna uma lista de itens sugeridos para completar o código.

Os itens da sugestão tem retornos bem diversificados, porém retornam frequentemente (i) um rótulo para descrever o item; (ii) a operação para completar o item, se aceito (ex., inserir/substituir o texto); (iii) o novo texto; e (iii) o intervalo que será alterado (linha e intervalo de colunas). Uma requisição adicional é feita para aceitar o item selecionado pelo usuário e efetivamente alterar o código (Figura 14).

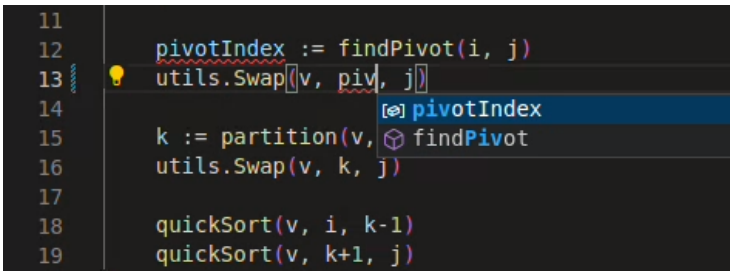

Figura 14 – Exemplo de utilização da funcionalidade `Completion`.

O usuário aciona as sugestões de código (comumente através do atalho de teclado Ctrl+Espaço). O cliente então requisita ao servidor a funcionalidade informando a posição do cursor (linha 13 e coluna 18 do arquivo quick.go). O servidor retorna uma lista ordenada com as opções para completar o símbolo.

Find References (*Code Comprehension*, 24 servidores): coleta todas as referências que apontam para o símbolo na posição desejada. O cliente requisita a funcionalidade para o servidor informando (i) a identificação do arquivo; e (ii) a posição do cursor (linha e coluna) (Figura 15).

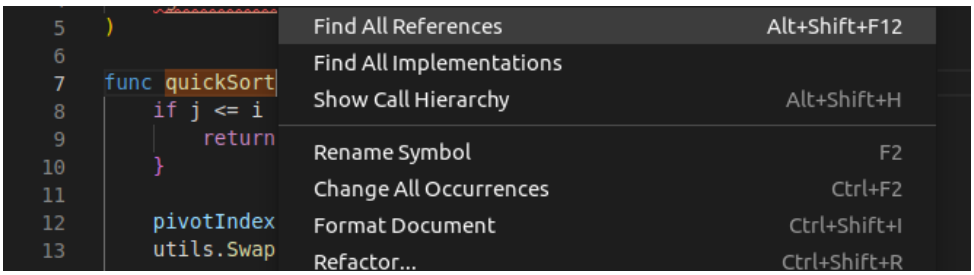

Figura 15 – Exemplo de utilização da funcionalidade `Find References` - Disparo

O usuário requisita ao cliente onde estão localizadas as referências para a função *quickSort* (ex., onde é chamada). O cliente então requisita ao servidor a funcionalidade informando a posição do cursor (linha 7 e coluna 14 do arquivo quick.go).

O retorno contém um conjunto com as localizações onde o símbolo é referenciado. Todos os itens do conjunto contém (i) a identificação do arquivo; e (ii) e posição (linha e coluna), que podem ser utilizados pelo cliente para navegar ou exibir essas referências (Figura 16).

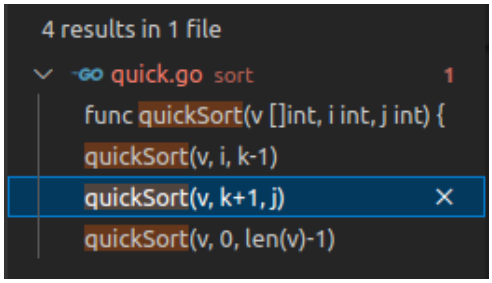

Figura 16 – Exemplo de utilização da funcionalidade `Find References` - Resultado

O servidor retorna então uma lista contendo as localizações onde a função foi utilizada. O cliente define a melhor forma de exibir os itens da lista e provê formas de navegar pelos itens ou mostrar prévias de código.

Document Symbols (*Code Comprehension*, 23 servidores): retorna a coleção de símbolos presentes em um documento. Os símbolos podem conter (i) informações como tipos, estado de depreciação ou localizações ou (ii) uma estrutura com uma hierarquia de símbolos (MICROSOFT, 2022b).

O corpo da requisição é mais simples para essa funcionalidade, necessitando apenas do identificador do documento. A resposta contém todos os símbolos identificados no documento. Cada símbolo retornado na lista deve conter ao menos (i) seu tipo (ex., variáveis, funções, classes), (ii) o nome do símbolo, e (iii) o intervalo onde foi declarado (linhas e colunas) (Figura 17).

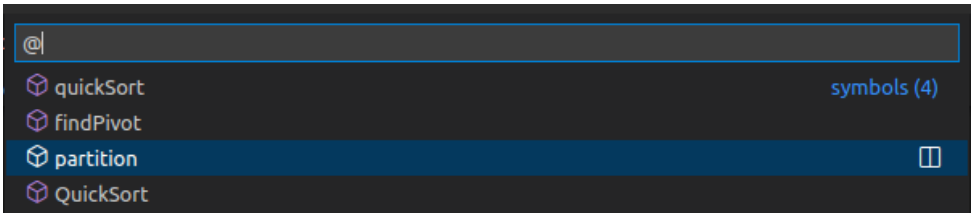

Figura 17 – Exemplo de utilização da funcionalidade `Document Symbols`

No editor *Visual Studio Code*, o usuário pode utilizar um atalho de teclado para requisitar a lista de símbolos no documento aberto. O cliente então envia ao servidor a identificação do arquivo, no exemplo o arquivo é o quick.go. O servidor retorna a lista de símbolos e o cliente decide a melhor forma de apresentar a lista e navegar entre as opções.

Signature Help (*Code Comprehension*, 20 servidores): prove informações de assinatura para uma dada posição do cursor. Ao solicitar as informações da assinatura o cliente informa ao servidor (i) o identificador do documento e (ii) posição do cursor (linha e coluna). O servidor identifica o contexto onde o cursor se aplica e retorna, quando uma chamada de função/método, qual a assinatura da mesma e o parâmetro selecionado (Figura 18).

Ao investigar as sete funcionalidades mais populares, foi identificada que a razão para a sua popularidade pode ser o fato de serem as mais genéricas da lista presente na Figura 9, ou seja, são independentes de paradigma de programação. Essas funcionalidades

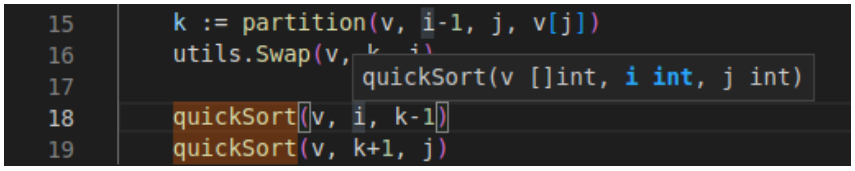

Figura 18 – Exemplo de utilização da funcionalidade `Signature Helper`

No editor *Visual Studio Code*, o usuário pode utilizar um atalho de teclado para requisitar a descrição da assinatura do método atrelado ao cursor. O cliente envia ao servidor o identificado do arquivo, no exemplo o arquivo é o quick.go, bem como a posição do cursor (linha 18 e coluna 15). O servidor retorna a assinatura do método e qual a variável (utilizando o nome definido na declaração do método) que está selecionada (no exemplo a variável *i* do tipo *int*.

cobrem aspectos variados do suporte à linguagem, indicando que o suporte à edição não depende primariamente de um tipo especifico de funcionalidade, mas ao invés de um conjunto diverso. A seguir será apresentado quão heterogênea é a cobertura do suporte à edição pelas funcionalidades mais populares de acordo com uma categorização informal do tipo de operação realizada:

- **Resolução de referências:** `Goto Definition` e `Find References`
- **Navegação:** `Goto Definition` e `Find References` e `Document Symbols`
- **Agregação de informações relevantes:** `Hover`, `Document Symbols` e `Signature Help`
- **Refatorações:** `Completion`
- **Feedback de código:** `Diagnostics`

Servidores distintos em quantidade de funcionalidades: Apesar da popularidade dessas funcionalidades, sendo altamente cobertas pela maioria dos servidores, se destacam três servidores que implementam somente algumas poucas delas (#8, #24 e #28). Ao investigar as razões para isso, descobre-se que esses servidores utilizam o LSP somente como base para oferecer funcionalidades não especificadas pelo protocolo, servindo assim como *plugins* (Tabela 4). Uma característica em comum entre esses servidores, que não tem como propósito geral prover um completo suporte à edição a linguagens, é que eles geralmente delegam as funcionalidades para bibliotecas externas que se limitam a somente as funções necessárias para o propósito desejado.

Casos distintos de funcionalidades não implementadas: Enquanto os números observados nos permitem estimar a popularidade de diferentes funcionalidades, eles estão suscetíveis a mudanças. A propósito, a funcionalidade `Hover` não estava disponível no *Robot Framework* (#11) no momento da análise, porém foi adicionada em versões posteriores a coleta de dados, aumentando a importância dessa funcionalidade. Para a

Tabela 4 – Servidores distintos que implementam poucas funcionalidades

Servidor	**Descrição**
SonarLint (#24)	O servidor não implementa nenhuma funcionalidade além da `Execute Command`, que invoca o SonarLint (SonarSource, 2022) para prover resultados de análise estática de código.
nokia/ntt (#28)	É um servidor LSP para testes agnósticos com o TTCN-3, que é uma linguagem específica de domínio para criar scripts de testes.
Turtle (#8)	É base para a IDE Stardog (Stardog Union, 2022), somente implementa as funcionalidades `Folding Range`, `Hover` e `Diagnostics` para uma ferramenta de exploração de dados.

funcionalidade `Find References`, o servidor *Vala* (#30) tinha uma requisição de implementação em aberto no seu repositório durante a análise que foi sinalizada como finalizada nesse intervalo. Todavia, pode-se assumir que, as funcionalidades mais populares serão implementadas primeiro e podem ser utilizadas para o design dos componentes reutilizáveis do projeto.

4.2.2 Relação entre funcionalidades e linguagens

Anteriormente, as funcionalidades de edição mais populares foram investigadas e foram discutidas as possíveis razões para tal. Apesar de serem conhecidas quais as funcionalidades que podem ser implementadas por um servidor LSP padrão, devido a limitações de recursos, isso não é informação o suficiente para desenvolver um servidor LSP. Decidir quais funcionalidades implementar é outro questionamento possível para quem pretende criar um novo servidor. Dado esse problema, foram investigados dois possíveis fatores de influência. Primeiro, se as características das linguagens podem impactar a viabilidade de diferentes funcionalidades. Segundo, se há interação entre as funcionalidades (ex., funcionalidades que se complementam).

A relação entre as linguagens e a funcionalidades implementadas. Para todos os servidores da amostra, foi calculada a correlação entre as funcionalidades implementadas e as caraterísticas da linguagem suportada. Utilizando para isso a implementação em R (RDocumentation, 2022) do método de *Pearson* (PEARSON, 1896). O resultado dessa análise é demonstrado na Figura 19.

Como características de linguagem, foram considerados tanto o tipo da linguagem (GPL ou DSL) quanto os paradigmas de linguagem (ex., Imperativa, Declarativa ou Funcional). Primeiro foi observado que as GPLs tendem a ter uma correlação positiva com todas as funcionalidades do LSP enquanto as DSLs tendem a uma correlação negativa, significando que os servidores das GPLs implementam mais funcionalidades do que aqueles

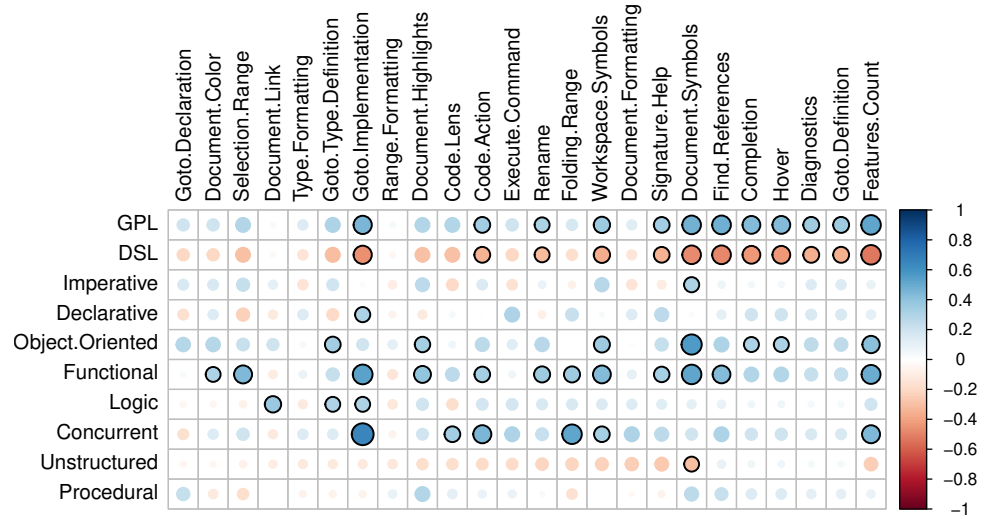

Figura 19 – Correlação entre funcionalidades de edição e as características das linguagens.

No eixo vertical estão as características das linguagens suportadas. No eixo horizontal as funcionalidades. Na última coluna a correlação baseada na quantidade de funcionalidades implementadas pelo servidor. Baseado no tamanho de nossa amostra, todas as correlações com uma correlação absoluta maior que 0.306 são estatisticamente significantes em um nível de 5% (LEE, 2005) e estão destacadas com um círculo ao seu redor.

para DSLs. Um efeito dessa correlação pode ser observado também na Figura 9, onde a lista está ordenada por quantidade de funcionalidades implementadas e as DSLs estão localizadas no topo. Apesar dessa observação, DSLs populares como XML, Latex ou Xtext ainda implementam diversas funcionalidades. Portanto, o número de funcionalidades implementadas pode depender mais da popularidade da linguagem do que do seu uso já que essas linguagens tendem a terem maiores comunidades suportando suas ferramentas, ou seja, mais recursos para implementar o suporte à edição, mesmo as funcionalidades menos relevantes. Além das DSLs, a popularidade como fator de influência para a quantidade de funcionalidades implementadas também é evidenciado pelo fato de linguagens populares de programação geral como Java, C/C++ e Rust serem as que implementam mais funcionalidades.

Ao observar as correlações individuais entre tipos de linguagem e as funcionalidades, percebe-se que a funcionalidade `Goto Implementation` tem uma correlação forte com as linguagens *Funcionais* e *Concorrentes*. Na prática, existe uma distinção explicita entre a assinatura e implementação para essas linguagens, através de interfaces ou protótipos, explicando a popularidade observada. Apesar de se esperar que qualquer linguagem *Orientada a Objetos* suporte essa funcionalidade, ela só é facilmente possível para linguagens baseadas em classes (ex., Java). Também pode-se observar que linguagens *Concorrentes*, característica comum em linguagens OOP populares do mercado como Java e C#, tendem a implementar mais funcionalidades.

Linguagens *Orientadas a Objetos* e *Funcionais* tem uma forte correlação com a

funcionalidade `Document Symbols`, refletindo a forte estruturação de tais linguagens e a necessidade de assistir os desenvolvedores provendo todos os diversos símbolos que estão contidos em um documento. Assim sendo, não somente a popularidade, mas a complexidade e a quantidade de recursos disponibilizados pela linguagem são fatores determinantes na quantidade e variedade de funcionalidades implementadas.

A única correlação negativa encontrada e estatisticamente significante entre as características da linguagem e as funcionalidades implementadas foi na implementação de `Document Symbols` que tende a não ser implementada em linguagens *Não estruturadas* (ex., Assemblers). Devido as serem compostas de estruturas de nível mais baixo, essas informações sobre os símbolos podem ser difíceis de coletar ou até mesmo não estar presente para essas linguagens.

Para as demais funcionalidades não foram encontradas correlações significantes na amostra, tanto por elas serem implementadas por praticamente todos os servidores LSP ou somente por alguns. Em resumo, apesar de todas as funcionalidades serem passíveis de implementação para aproximadamente todos os servidores LSP, existem suportes a edição que são específicos para determinados paradigmas de linguagem.

Correlação entre as funcionalidades existentes. Para identificar as funcionalidades que são frequentemente implementadas em conjunto, foi utilizada a mineração de regras de associação, que demonstra associações entre um conjunto de itens de dados (AGRAWAL; SRIKANT et al., 1994), demonstrando por exemplo relações do tipo:

$$A \Rightarrow B$$

A implica, ou é mais provável de ser implementada, em conjunto com B.

Para esse propósito, foi utilizado o algoritmo Apriori, disponível no pacote R *arules*[2]. Como parâmetro, o limiar de *suporte* foi setado para 0.63, o que significa que garantimos que as funcionalidades presentes em pelo menos 63% (aproximadamente dois terços) dos servidores seriam avaliadas, e *confiança* de 0.95, o que significa que as funcionalidades devem aparecer em conjunto pelo menos em 95% dos casos.

O algoritmo minerou 167 regras de associação, que são sumarizada na Figura 20. Essas regras de associação demonstram quando combinações de funcionalidades são encontradas em conjunto. As posições 1 a 5 indicam quantas funcionalidades estão classificadas como *antecedentes*, o *rhs* indica os *consequentes*. A espessura das setas representa o *suporte*, mais espesso significa mais forte, e quão forte o *sombreamento* indica quão possível é a associação de ocorrer. Ao analisar a figura, pode-se observar que:

[2] <https://cran.r-project.org/web/packages/arules/index.html>

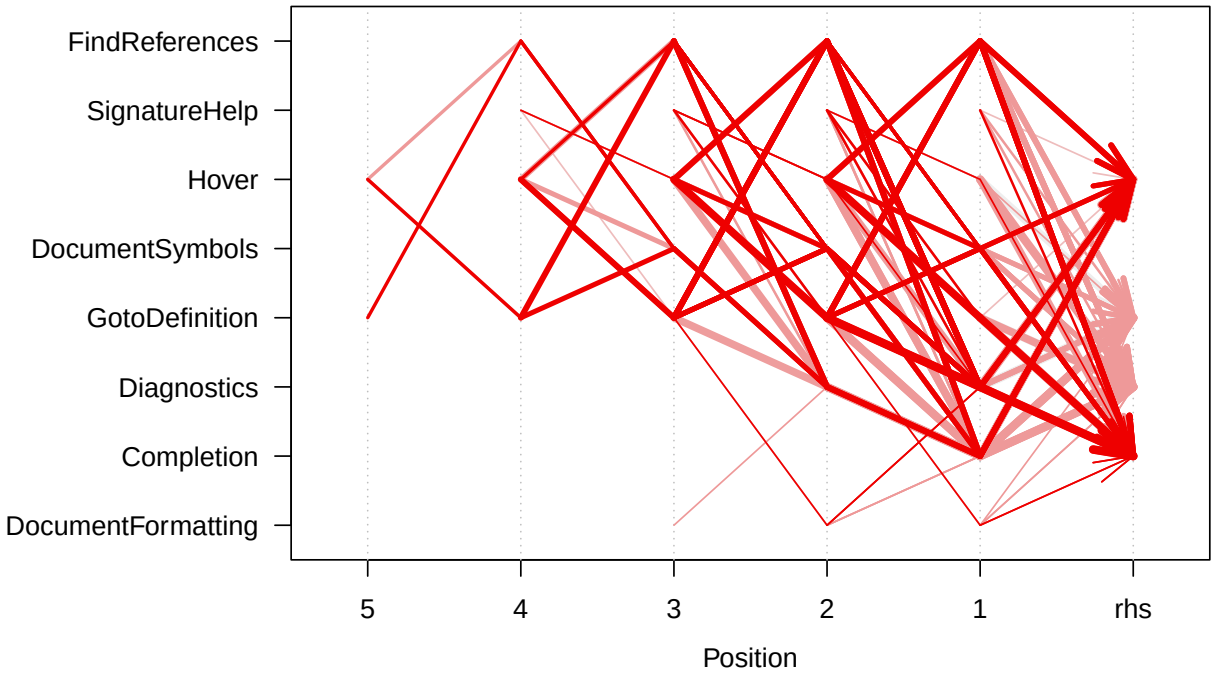

Figura 20 – Coordenação paralela de 167 regras de associação entre as funcionalidades mais populares

- Há algumas regras com 5 ou 4 funcionalidades, envolvendo principalmente `Goto Definition` e `Hover`, que são as primeiras e terceiras funcionalidades mais populares;
- Não há regras em que as funcionalidades consequentes são `Find References`, `Signature Help`, `Document Symbols` e `Document Formatting`;
- As regras com sombreamento mais fortes como consequentes são `Hover` e `Completion`;
- `Find References`, `Diagnostics` e `Completion` estão entre as funcionalidades com *suporte* mais forte.

No geral, esses resultados demonstram que, quando os servidores implementam as funcionalidades populares `Find References`, `Diagnostics` e `Goto Definition` elas terão mais chances de implementar as funcionalidades `Hover` e `Completion`. Ou seja, implementando as três primeiras funcionalidades, há um potencial de reúso para então implementar as demais, uma hipótese é a reutilização de componentes em comum (para representação ou travessia de código). No caso da funcionalidade `Diagnostics`, o fato da mesma coletar as informações necessárias durante a compilação do código, implica na implementação de outras funcionalidades, pois a compilação é uma pré-condição para a maioria das demais funções.

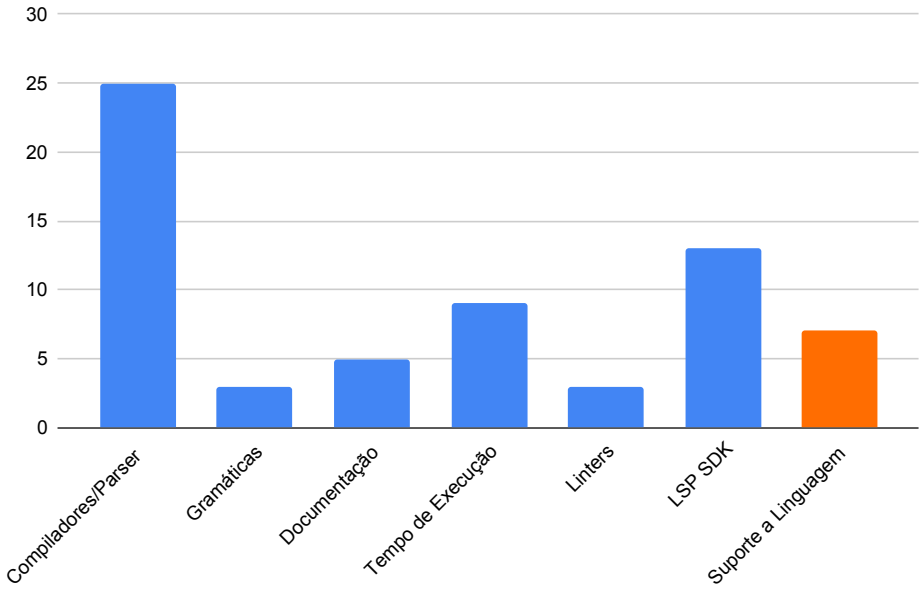

Figura 21 – Gráfico da frequência em que as bibliotecas são reutilizadas pelos servidores da amostra

Os valores do eixo Y representam a quantidade de servidores que reutilizam essa categoria de biblioteca. Em azul as Bibliotecas utilizadas comumente e discutidas no Capítulo 4.3.1. Em laranjado as Bibliotecas de Suporte à Linguagem discutidas no Capítulo 4.3.2.

4.3 Aspecto 2: Utilização de bibliotecas de terceiros

Bibliotecas são uma prática comum na programação para criar e compartilhar componentes reutilizáveis de software (GRISS, 1993). Portanto, com interesse nos papéis que as bibliotecas externas tem na implementação de LSPs, esse aspecto trata dos tipos recorrentes de bibliotecas que suportam as implementações dos servidores da amostra. Primeiro, foram identificadas as bibliotecas utilizadas frequentemente para atividades pontuais e em seguida as que encapsulam completamente o suporte à edição.

4.3.1 Bibliotecas utilizadas comumente

Coletar e processar informações a partir da representação do código-fonte são tarefas necessárias para a implementação de funcionalidades LSP. Para esse e outros problemas recorrentes, bibliotecas disponibilizam soluções maduras que podem ser reutilizadas. Foram identificados seis diferentes tipos de bibliotecas que são frequentemente utilizadas na implementação de servidores LSP (Figura 21).

Compiladores/*Parsers* (25 servidores): A utilização de compiladores ou *parsers* é altamente comum. Essa grande frequência de utilização é esperada, pois essa categoria de bibliotecas é utilizada principalmente para compilar a instância do código-fonte em uma representação abstrata, informação necessária para a eficiente exploração

das instâncias de programação de um documento. Compiladores/*parsers* também provêm erros/avisos de problemas de sintaxe encontrados no código-fonte. Em um nível mais alto de abstração, essas bibliotecas fornecem estratégias reutilizáveis para travessia das instâncias do código-fonte. Um exemplo é o *Tolerant PHP Parser* (MICROSOFT, 2022a), outros compiladores identificados podem ser encontrados na Tabela 5.

Tabela 5 – Relação de Compiladores/Parser identificados na amostra

Servidor	**Compilador/Parser**	**Repositório/Docs**
#01	Apache Royale	<github.com/apache/royale-compiler>
#02	xmlparsedata	<github.com/r-lib/xmlparsedata>
#03	Go/buildTools	<github.com/golang/tools>
#04	Erlang OTP	<github.com/erlang/otp>
#05	flux::parser	<github.com/influxdata/flux>
#06	hlasm	NF/C
#07	Apache Xerces	<svn.apache.org/repos/asf/xerces/java>
#08	stardog-union/millan	<github.com/stardog-union/millan>
#09	llvm/clang	<github.com/llvm/llvm-project>
#10	Eclipse JDT	<github.com/eclipse-jdt>
#11	Robot Framework API	<github.com/robotframework/robotframework>
#12	Hir Semantics	<github.com/rust-lang/rustc-dev-guide>
#13	Eclipse EMF	<eclipse.org/modeling/emf>
#17	Puppet::Pops::Parser	<github.com/puppetlabs/puppet>
#18	microsoft/tolerant-php-parser	<github.com/microsoft/tolerant-php-parser>
#19	Groovy Compiler	<github.com/groovy/groovy-core>
#20	cstree	<github.com/domenicquirl/cstree>
#21	Camel Parser	<github.com/apache/camel>
#22	Ballerina Lang	<github.com/ballerina-platform/ballerina-lang>
#23	Java Compiler	<github.com/openjdk>
#25	Lua Lexer	NF/C
#26	Moca Lexer	NF/C
#27	Multiple	Baseado no ambiente
#28	ttcn3 Parser	<github.com/nokia/ntt>
#30	libvala:valac	<gitlab.gnome.org/GNOME/vala/>

NF/C = Não encontrado (normalmente por ser biblioteca com código-fonte fechado) ou implementação embarcada no próprio projeto do servidor

Grammars (3 servidores): Gramáticas de linguagem são formas de expressar o código-fonte através de uma sintaxe independente do contexto. Essa gramática é então utilizada para gerar os *tokens*, e então um parser que identifica esses *tokens* a partir do código-fonte. Uma ferramenta comum para criar *parsers* a partir de gramáticas de linguagem é o Antlr (PARR, 2022). Uma vez com o parser gerado é possível para o mesmo interpretar o código-fonte/modelo e então criar uma representação abstrata do seu conteúdo.

Além de gerar o parser, o servidor LSP tem a função adicional de manter a gramática, por exemplo, ao surgirem estruturas novas ou sendo elas depreciadas dado novas versões da linguagem de programação mantida. Um exemplo de gramática embutida e mantida no projeto, é no servidor *MOCA* (#26). Outros exemplos podem ser encontrados na Tabela 6. O servidor #7 foi classificado como usuário de gramática dado a sua integração com XSD, permitindo utilizar em tempo de execução gramáticas definidas pelo usuário para validar diferentes esquemas de dados no XML avaliado. Apesar de utilizar gramáticas, o servidor *Lua* (#25) também implementa o parser para identificar os *tokens*, uma implementação mais complexa de parser mantido pelo próprio servidor, não considerado nessa categoria.

Tabela 6 – Relação de Gramáticas de Linguagem identificados na amostra

Servidor	**Tipo de Gramática**	**Gramática de exemplo**
#07	XSD	NF/E
#15	Antlr	<github.com/eclipse/che-che4z-lsp-for-cobol/blob/1e54e7a73f7450444ba4afae82570cdb4f7d3158/server/src/main/antlr4/org/eclipse/lsp/cobol/core/parser/CobolLexer.g4>
#26	Antlr	<github.com/mrglassdanny/moca-language-server/blob/master/src/main/antlr/Moca.g4>

NF/C = Não encontrado (somente para o caso da utilização de XSD no servidor #7)

Manipulador de Documentação (5 servidores): Especificações de documentação como a JavaDoc são geralmente declaradas em outra sintaxe, que não a instância da linguagem suportada. Comumente essas documentações são ignoradas pelos compiladores e *parsers*, podendo ser mesmo armazenadas em diferentes documentos. Bibliotecas como a Jedi (HALTER, 2022) (Python), ElixirSense (Elixir Language Server Protocol, 2022) (Elixir) e DocBlock (PHPDOCUMENTOR, 2022) (PHP), podem lidar com tais documentações e mapeá-las para as instâncias de elementos correspondentes no código-fonte. Essas são bibliotecas que simplificam a implementação para os desenvolvedores de suporte à linguagem, que não necessitam manipular texto para identificar essas informações no código-fonte ou em arquivos adicionais. Outros exemplos podem ser encontrados na Tabela 7.

Tabela 7 – Relação de Manipuladores de Documentação identificados na amostra

Servidor	**Ferramenta**
#02	<r-project.org/help.html>
#14	<github.com/davidhalter/jedi>
#16	<github.com/msaraiva/elixir_sense>
#18	<github.com/phpdocumentor/phpdocumentor>
#22	<github.com/ballerina-platform/ballerina-lang>

Manipuladores de informações de ambiente de execução (9 servidores): Além de informações das instâncias do código-fonte e de especificações da linguagem, informações específicas do ambiente podem ser necessárias para implementar as funcionalidades. Um exemplo, identificar versões da linguagem de programação instaladas no ambiente de desenvolvimento ou mesmo, identificação de bibliotecas instaladas no ambiente através de ferramentas de *build*. Essas informações podem ser utilizadas na implementação de funcionalidades como `Code Completion`. Como exemplos dessa categoria pode-se citar a M2Eclipse (Eclipse Foundation, 2022d) e Buildship (Eclipse Foundation, 2022a) que são utilizadas para coletar informações de dependências gerenciadas pelos empacotadores Maven ou Gradle em projetos Java. Exemplos adicionais na Tabela 8

Tabela 8 – Relação de Manipuladores de Informações de Ambiente de Execução identificados na amostra

Servidor	Ferramenta	Necessidade
#02	Funções da própria linguagem R	Obter pacotes instalados e suas descrições
#03	Sistema de reflexão provido pela linguagem Go	Obter detalhes das instâncias de código
#04	<github.com/erlang/rebar3>	Obter detalhes do compilador e informações do ambiente
#05	Funções da *stdlib* do Flux	Coletar nomes dos pacotes disponíveis
#10	<github.com/eclipse/buildship>; <eclipse.org/m2e/>	Manipular e coletar informações sobre as bibliotecas utilizadas no projeto
#14	<github.com/davidhalter/jedi>	Gerenciar os diferentes ambientes Python disponíveis
#17	<sarti.dev/puppetfile-resolver>	Resolução de dependências de módulos do Puppet
#19	API de reflexão do Java/Groovy	Obter detalhes das instâncias de código
#21	<camel.apache.org/camel-k/1.7.x/architecture/cr/camel-catalog.html>	Coletar metadados do ambiente de execução

Linter (3 servidores): A funcionalidade `Diagnostics` do LSP disponibiliza informações como erros de sintaxe reportadas por um compilador. Adicionalmente, *code linters* ou outras ferramentas de análise estática/dinâmica de código podem ser utilizadas pelos servidores para coletar diagnósticos adicionais. Um exemplo é a biblioteca *pylint* (Python Code Quality Authority, 2022) que é executada baseada em um temporizador pelo servidor #14, validando os arquivos e retornando avisos relacionados aos mesmos como: (i) a formatação apropriada do código, (ii) *smells* presentes, ou (iii) refatorações recomendadas. Os servidores *Elixir* (#16) e *Erlang* (#4) utilizam a ferramenta de análise estática Dialyzer (Ericsson AB, 2022) para prover o mesmo tipo de diagnóstico adicional.

LSP SDK (13 servidores): Os detalhes de implementação do LSP envolvem informações como (i) diferentes esquemas de mensagem para requisição e respostas para inicializar o servidor ou executar as funcionalidades; (ii) o protocolo de comunicação baseado em chamadas remotas; (iii) requisições partindo do servidor; dentre outros. Implementar todos esses detalhes requer um esforço considerável por parte dos engenheiros. Contudo, existem Kits de Desenvolvimento (SDKs) que podem auxiliá-los. Foram encontradas múltiplas utilizações do *SDK LSP4J* (Eclipse Foundation, 2022c) para servidores implementados com a linguagem *Java* (11 servidores). Bem como uma utilização do *Microsoft/vscode-languageserver-node* (Microsoft, 2022) para servidores implementados com as linguagens *Javascript/Typescript* e execução baseada no *node.js* [3]. Também foi encontrada uma utilização do SDK *gvls* no servidor *Vala* (#30).

Em resumo, bibliotecas são geralmente utilizadas em nossa amostra para evitar implementações customizadas ao acessar instâncias do código-fonte ou informações adicionais para implementar as funcionalidades do LSP.

4.3.2 Bibliotecas empacotadas de suporte à linguagem

Enquanto bibliotecas geralmente suportam a implementação de funcionalidades e seus detalhes, sete servidores se beneficiaram de bibliotecas que implementam as funcionalidades do protocolo totalmente ou parcialmente e podem ser encontradas na Tabela 9. Por exemplo, Eclipse JDT (Eclipse Foundation, 2022b) é uma biblioteca utilizada pelo servidor *Java* (#10) que implementa funcionalidades como `Completion` e `References`. O mesmo caso foi evidenciado com a utilização da biblioteca Jedi (HALTER, 2022) pelo servidor *Python* (#14). Essas bibliotecas provem funcionalidades o bastante para que o servidor se torne uma fachada (referenciando o padrão *Facade* de Gamma et al. (1995)) que as encapsula. A comunicação com essas bibliotecas costuma ser simples, o servidor deve somente informar a instância do código-fonte para a biblioteca, obtendo assim a resposta da mesma.

Empacotar uma biblioteca desse tipo não necessariamente significa que o servidor deve somente delegar todas as suas responsabilidades para elas, sem implementar nenhuma lógica. Esses servidores ainda precisam controlar o acesso ao código-fonte ou qualquer uma de suas representações, bem como sinalizar as bibliotecas que elas devem invalidar qualquer cache quando mudanças ocorrem. Outra responsabilidade usual que esses servidores precisam implementar é a chamada aos *Linters* que devem reanalisar o código após a ocorrência de mudanças.

Apesar da utilização dessas bibliotecas diminuir consideravelmente o esforço para implementar um servidor LSP, os engenheiros devem considerar o esforço extra para traduzir as interfaces e respostas esperadas no protocolo de/para o que é esperado por

[3] <https://nodejs.org/>

Tabela 9 – Relação de bibliotecas que provêm suporte à edição para os servidores LSP

Servidor	**Biblioteca**	**Repositório**
#10	Eclipse JDT	<eclipse.org/jdt>
#14	Jedi	<github.com/davidhalter/jedi>
#16	Elixir Sense	<github.com/elixir-lsp/elixir_sense>
#25	Intellij Psi	<plugins.jetbrains.com/docs/intellij/psi.html>
#27	Merlin	<github.com/ocaml/merlin>
#29	sourcekitd e Clang	<github.com/apple/swift/tree/main/tools/SourceKit/tools/sourcekitd>, <clang.llvm.org/>
#30	gvls	<github.com/esodan/vala-language-server>

cada biblioteca (ex., traduzindo as mensagens de diagnóstico dos *linters* para a interface esperada pela especificação LSP). Outra preocupação é evitar que as interfaces das bibliotecas interfiram ou mesmo poluam/substituam as interfaces da implementação do servidor, padrões como o *Adapter* ou *Facade* (GAMMA et al., 1995) devem ser levados em consideração.

4.4 Aspecto 3: Estruturação dos servidores LSP

A arquitetura de um sistema tem um papel essencial na engenharia de software, sendo crucial para o sucesso do desenvolvimento e manutenção de sistemas de software (GAMMA et al., 1995; BASS; CLEMENTS; KAZMAN, 2003). Na literatura, vários padrões arquiteturais (GAMMA et al., 1995) foram propostos, porém quando aplicados inadequadamente, anti padrões (BROWN et al., 1998; PELDSZUS et al., 2016) podem emergir, impedindo a adequada evolução de um sistema. Portanto, deve-se identificar quão apropriado são esse princípios de design para os servidores LSP.

Ao investigar o código-fonte dos servidores LSP, foram identificados dois princípios de design aplicados/aplicáveis na implementação de todos os servidores investigados. Primeiro, a arquitetura dos servidores LSP pode ser estruturada utilizando o estilo arquitetural de camadas. Segundo, os servidores LSP devem levar em consideração qual é o fluxo geral para implementar completamente uma funcionalidade ao estruturar a comunicação interna entre os componentes 4.1.

4.4.1 Arquitetura em camadas

Os servidores LSP da amostra, usualmente são estruturados em três camadas de responsabilidade. Cada uma dessas camadas representa um aspecto técnico da implementação que as funcionalidades de edição devem lidar. Uma visão geral dessas linguagens e

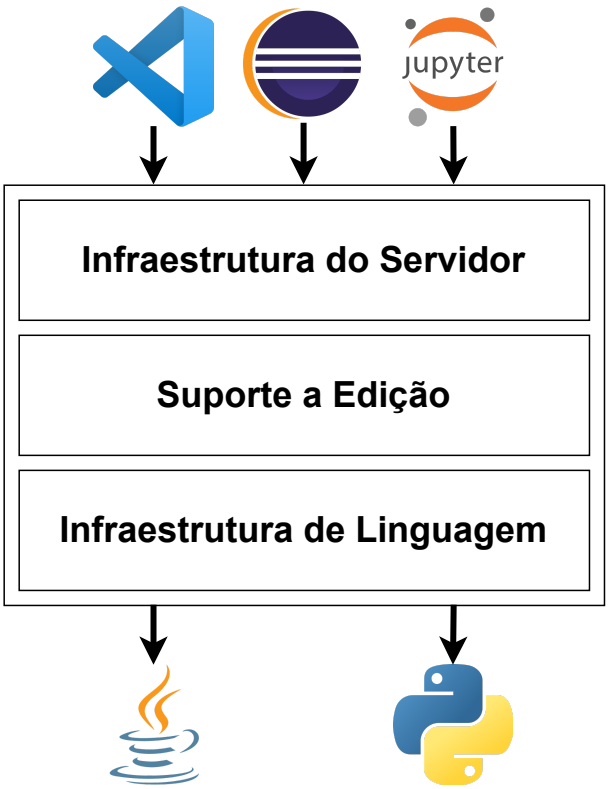

Figura 22 – Camadas básicas para a implementação de um servidor LSP

sua topologia pode ser encontrada na Figura 22. Considerando as responsabilidades, as camadas podem ser descritas da seguinte forma:

- **Infraestrutura do Servidor:** Essa camada é responsável por lidar com detalhes do protocolo como (i) esquemas de requisição e resposta; (ii) comunicação GRPc; e (iii) mensagens padrão de inicialização do servidor; dentre outros. A implementação dessa camada pode ser delegada para um SDK LSP. Uma lista de SDKs LSP para diferentes linguagens de programação pode ser encontrada na página oficial do LSP[4] e uma discussão sobre os exemplos observados pode ser encontrada na Seção 4.3.

- **Suporte à Edição:** Essa camada contém o conhecimento necessário para processar especificamente as funcionalidades. Por exemplo, coletando os dados e atravessando a representação abstrata do código-fonte para coletar os itens que serão sugeridos na funcionalidade `Completion`. Alguns servidores optam por delegar essa responsabilidade para alguma *Biblioteca de Suporte à Edição* que provê funcionalidades como `Rename` e `Completion`, mais detalhes na Seção 4.3.

- **Infraestrutura de Linguagem:** Essa é a estrutura de mais baixo nível e implementa a interação imediata com as instâncias de código-fonte (ex., *parsing* do código-fonte para criar a representação em uma sintaxe abstrata). Se necessário, o

[4] <https://microsoft.github.io/language-server-protocol/implementors/sdks/>

suporte para múltiplas linguagens pode ser provido dentro dessa camada. A entrada dessa camada é usualmente a instância do código-fonte e a saída é a instância de representação abstrata que pode ser processada eficientemente para implementar as funcionalidades do LSP (ex., tradução do código-fonte para uma AST). Como o processo de *parsing* pode gerar informações adicionais relevantes, como avisos/erros de sintaxe identificados, podem ser utilizados padrões como *Publish/Subscribe* (ou *Observers* como catalogado por Gamma et al. (1995)) para prover tais informações a camada de *Suporte à Edição* acima dela.

A figura 23 demonstra exemplos dessas camadas para três servidores LSP existentes. O servidor *Moca* (#26) é estruturado em pacotes bem definidos. O pacote *moca* contêm código para lidar com detalhes da *Infraestrutura da Linguagem* como o *parsing* e a sua representação abstrata, o pacote *services* contém a implementação do *Suporte à Edição* e a raiz do projeto contêm código para lidar com a *Infraestrutura do Servidor*.

Uma estrutura similar é seguida pelos servidor *Groovy* (#19), porém a diferença entre eles está no fluxo de informações, onde o *Moca* recebe a requisição e delega o processamento para a camada de *Suporte à Edição* que acessa as representações diretamente da camada de *Linguagem*. No caso do servidor *Groovy*, o servidor requisita as representações da camada de linguagem e então passa como parâmetro para o *Suporte à Edição*.

Porém nem todos os servidores seguem essa mesma estrutura. Por exemplo, *Lua* (#25) tem dois módulos principais: (i) *EmmyLua-Common* que agrega a *Infraestrutura de Linguagem* e o *Suporte à Edição* e (ii) *EmmyLua-LS* que provê a *Infraestrutura do Servidor*.

Mesmo que os servidores encapsulem o código em um mesmo nível de responsabilidade, o fluxo de informação pode ser divergente. Os fluxos mais comuns de execução serão discutidos a seguir.

4.4.2 Fluxo de execução das funcionalidades

Para servidores que implementam as funcionalidades, além da infraestrutura do LSP, foi identificado o fluxo de execução descrito na Seção 4.1. Depois de receber uma requisição e durante o processamento da funcionalidade, o servidor com o conteúdo do documento em memória chama para cada ação um provedor que lidará com ela. A execução das funcionalidades é geralmente composta por três etapas:

- ***Parsing* (interpretação):** Navegar através dos símbolos do código-fonte é geralmente requerido para implementar funcionalidades como a `Completion` e `References`. Portanto, a maioria dos servidores/funcionalidades analisados precisam analisar gra-

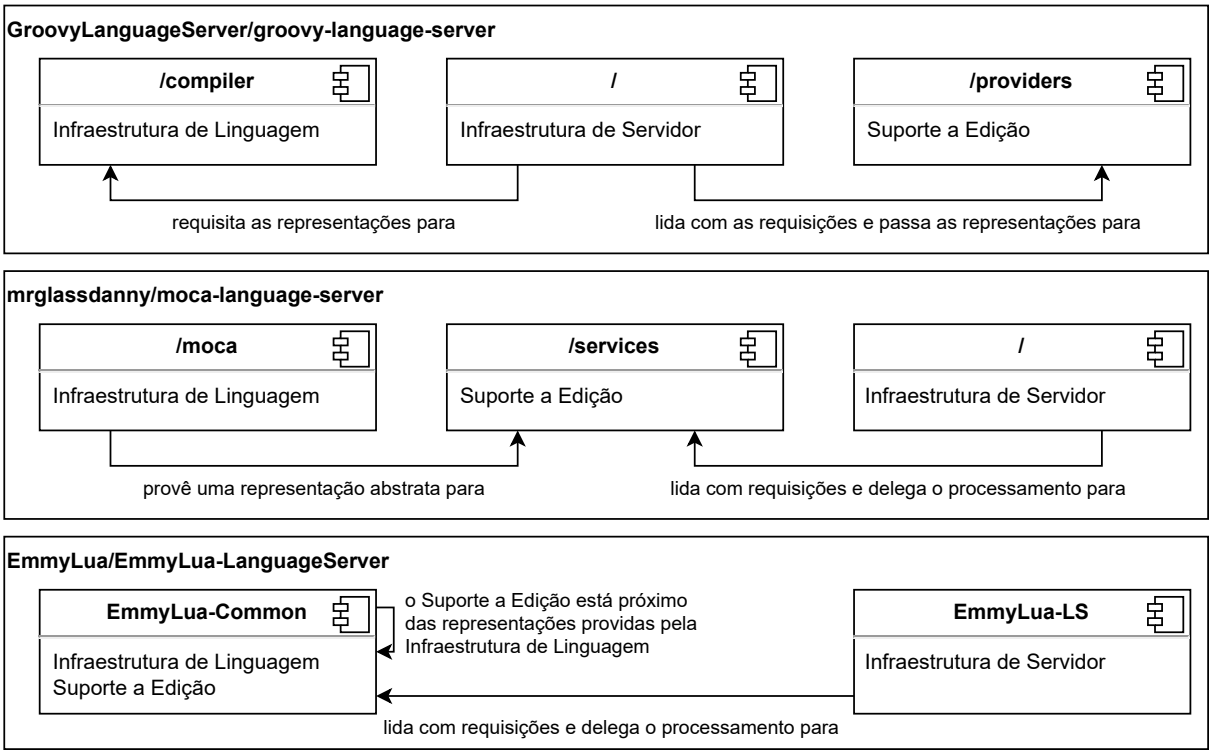

Figura 23 – Exemplos de camadas em servidores existentes

maticalmente o código-fonte para gerar uma representação abstrata em um primeiro passo de execução.

- ***Traverse* (travessia):** A sintaxe abstrata é então atravessada para identificar o conjunto de nós que serão necessários para processar as funcionalidades (ex., (i) coletar todos os nós que são símbolos em um arquivo fonte específico, ou (ii) identificar o nó que declara determinado símbolo).

- ***Process* (processamento):** Com o conjunto de nós relevantes coletados, a funcionalidade é efetivamente processada. Isso pode envolver filtrar/ordenar ou coletar informações adicionais dos nós que são relevantes para a resposta (ex., identificar qual alteração será necessária para substituir ou completar o símbolo atual referenciando outro nó, que será utilizado como sugestão de código).

As etapas de *Parsing* e *Traverse* podem ser executadas em diferentes momentos, como na inicialização do servidor, sendo assim realizado o cache dos dados para serem reutilizados quando necessário, assunto discutido nas Seções 4.7 e 4.8.

Além desse fluxo ser a sequência mais comum, casos específicos onde a execução de uma funcionalidade é delegada para alguma biblioteca podem resultar em uma sequência diferente de passos. Um exemplo é a funcionalidade `Diagnostics`, que geralmente coleta os resultados/erros do processo de interpretação/compilação ou mesmo uma biblioteca

de lint. O servidor então, só precisa traduzir as mensagens em um formato de resposta especificado pelo LSP, tornando todos os outros passos de execução obsoletos.

O fluxo observado é aderente ao padrão *Pipes and Filters*. Porém, enquanto encontra-se indicações desse padrão, nenhum servidor o seguiu estritamente. Entretanto, as implementações de servidores LSP podem ser estruturadas efetivamente em camadas contendo filtros específicos para tarefas que podem ser intercambiadas (ex., para suportar múltiplas instâncias de linguagem).

4.5 Aspecto 4: Granularidade de implementação

Ao implementar um novo servidor LSP, deve-se decidir em qual nível de granularidade as funcionalidades serão implementadas, ou mesmo quando faz sentido separar as implementações das funcionalidades em diferentes módulos. Dado essa necessidade, foi investigada a implementação de cada servidor LSP e cada uma foi atrelada a um nível de granularidade entre 1-5 em uma Escala de *Likert*. Representando as implementações com granularidade mais grosseiras (1) até as mais refinadas (5).

- **1: Monólito Simples (1 servidor):** Essa granularidade abrange os servidores que não implementam nenhuma funcionalidade, ou até mesmo múltiplas funcionalidades em uma única entidade no mesmo nível de granularidade (ex., uma classe). Somente *SonarLint* (#24), que não implementa nenhuma funcionalidade, está nesse nível de granularidade.

- **2: *Delegator* por funcionalidade (6 servidores):** Nesse nível de granularidade um único arquivo é utilizado para chamar uma biblioteca que irá efetivamente implementar a funcionalidade. Este é o caso para a maioria dos servidores que reutilizam *Bibliotecas de Suporte à Edição* como o servidor *Python* (#14). O único servidor que utiliza uma biblioteca de suporte mas não está nesse nível de granularidade é o *Swift* (#29), que além de delegar a execução para múltiplas bibliotecas externas, também implementa uma estrutura para suportar múltiplas linguagens (nível 5).

- **3: Arquivo/Classe por funcionalidade (13 servidores):** O caso mais comum é o de um servidor que utiliza um arquivo ou classe por funcionalidade, encapsulando métodos ou funções para implementar detalhes das funcionalidades (ex., estratégias de travessia ou funções específicas para validar o contexto onde um símbolo é aplicado). Um exemplo de organização de arquivos com esse nível de granularidade pode ser encontrado na Figura 24.

- **4: Módulo ou múltiplos Arquivos/Classes por funcionalidade (8 servidores):** Os detalhes de implementação de cada funcionalidade estão bem encapsulado

em múltiplos arquivos/classes. Esses arquivos irão em sua maioria navegar a representação abstrata buscando por nós específicos ou provendo contexto para selecionar nós alvo. Um exemplo seria a utilização de diferentes classes para coletar variáveis ou funções e assim gerar as sugestões de código no servidor *Ballerina* #22. Outra utilização existente é a de diferentes implementações para coletar *snippets/keywords* que serão utilizados nas sugestões de código do servidor *Robot* #11.

- **5: Múltiplos módulos por linguagem suportada (2 servidores):** Nesse nível de granularidade, cada funcionalidade é implementada em múltiplos módulos, por exemplo em servidores que atendem múltiplas linguagens. O servidor *Turtle* (#8) é uma das linguagens suportadas pela IDE *Stardog*, todas as linguagens suportadas pela ferramenta estendem a classe *AbstractLanguageServer* que orquestra a execução de funcionalidades. Cada implementação desse servidor abstrato é implementado em um módulo que deve prover métodos para carregar e realizar o *parsing* das instâncias de código-fonte de sua linguagem, bem como prover diagnósticos quando mudanças ocorrem nos documentos. O servidor #29 é um exemplo de servidor que contém estruturas refinadas para delegar a execução de uma funcionalidade para múltiplas *Bibliotecas de Suporte à Edição*, específicas para cada linguagem suportada (ex., a linguagem de programação *Swift*). Após a execução das funcionalidades pela biblioteca, o resultado é traduzida para a abstração definida pelo protocolo LSP (ex., uma resposta seguindo o esquema *HoverResponse*).

Observando a Figura 25, nota-se que os servidores tendem a utilizar implementações diretas, onde o nível 3, com somente um arquivo por funcionalidade é a granularidade mais comum. Há uma leve tendência em se utilizar estruturas mais refinadas e implementações mais complexas, com múltiplas classes por funcionalidade ou suportando múltiplas linguagens, porém a diferença na contagem de servidores é discreta em comparação com as implementações mais simples.

4.6 Aspecto 5: Representação de instâncias de linguagem

Para prover o suporte à edição, é necessária uma representação abstrata da instância do código-fonte, permitindo assim a navegação entre os símbolos programaticamente. Enquanto, em casos mais simples, os servidores LSP podem operar diretamente sobre a representação concreta do código-fonte, para a maioria das tarefas uma representação estruturada é necessária.

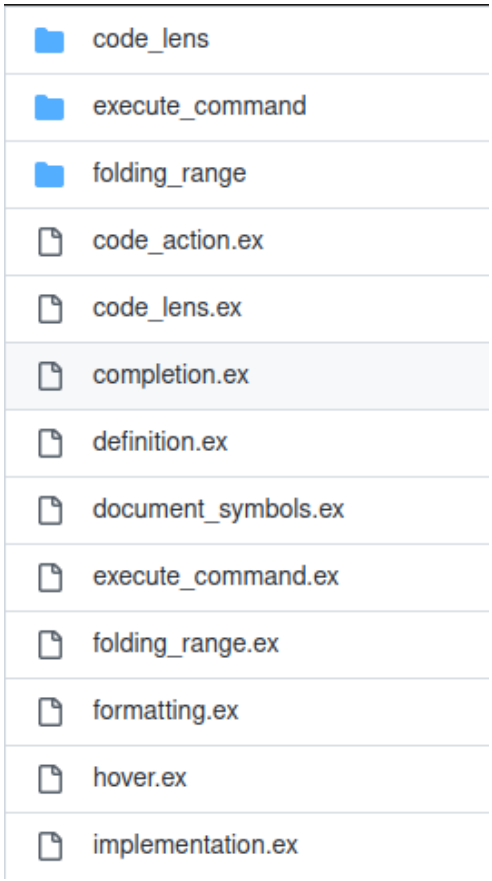

Figura 24 – Exemplo de organização de repositórios do servidor Elixir (#16)

4.6.1 Árvores Abstratas de Sintaxe

Para todos os 24 servidores que necessitam de uma estrutura de dados para representar as instância do código-fonte, as Árvores Abstratas de Sintaxe (AST) são utilizadas. Uma AST é uma representação estruturada da sintaxe concreta e baseada em árvores, onde cada nó da árvore representa os símbolos e as arestas representam as relações entre os símbolos (ver Seção 2.3). A AST pode ser navegada através de algoritmos de navegação para grafos ou árvores (CORMEN et al., 2022) e assim coletar informações necessárias sobre os símbolos contidos nela.

A estrutura de dados é criada através do processo de *parsing* assim que o cliente notifica para o servidor que o documento foi aberto (como no servidor *XML* #7). Outra opção para gerar as representações é, durante a fase de inicialização do servidor, todos os arquivos do espaço de trabalho serem carregados, compilados e então armazenados em cache (ver Seção 4.7). Os únicos casos onde a AST não é utilizada para representar as instância de linguagem são em servidores que utilizam *Bibliotecas empacotadas de suporte à linguagem* (ver Seção 4.3.2), pois essas bibliotecas já criam e manipulam as representações abstratas necessárias de forma encapsulada em seu código-fonte, retirando essa responsabilidade da implementação do servidor.

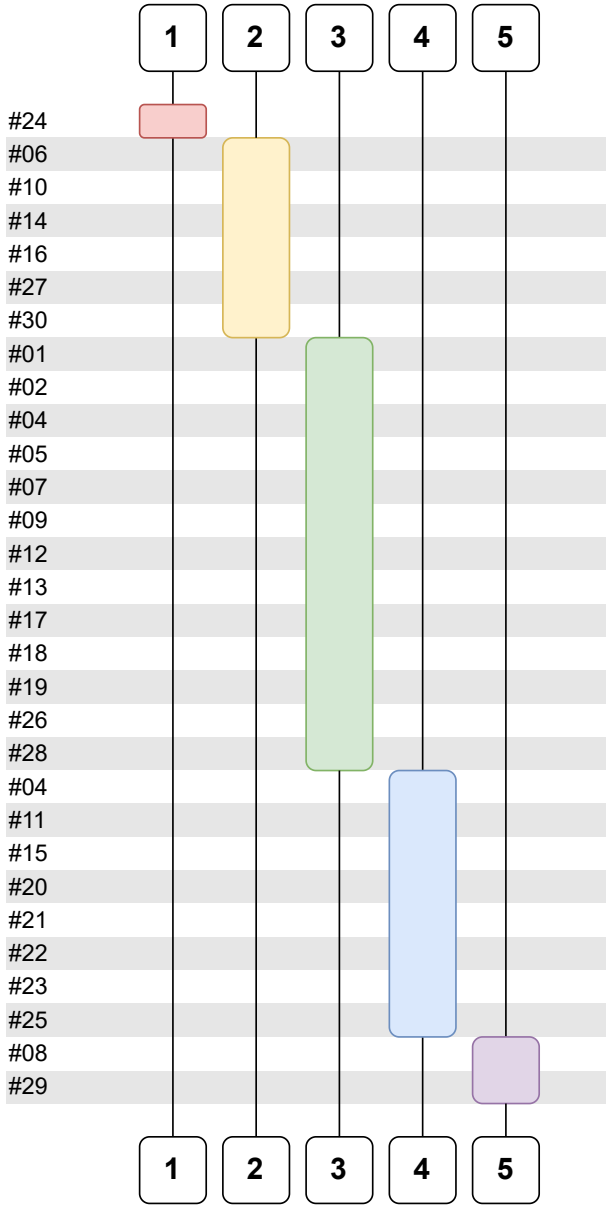

Figura 25 – Escala de Likert da granularidade das implementações dos servidores analisados

4.6.2 Travessia de árvores

O ato de atravessar a árvore para coletar informações é chamada de *Traversal* (LÄMMEL; VISSER; VISSER, 2003). Várias são os motivos para atravessar os símbolos de uma árvore, como: (i) identificar um símbolo em determinado *offset* (contagem de caracteres desde o inicio do documento); (ii) coletar os parâmetros de uma função; ou (iii) coletar todas as variáveis declaradas em um documento. A travessia da representação em árvore de código-fonte se torna então uma atividade obrigatória para a implementação das funcionalidades, caso o servidor se comprometa em implementar a camada de *Suporte à Edição* (ver Seção 4.4.1). Essa travessia pode ocorrer de quatro diferentes formas, identificadas durante a análise da nossa amostra:

Algoritmos padrão para Árvores/Grafos (13 servidores): Como as ASTs são geralmente instâncias de Árvores ou Grafos, a solução mais comum é atravessá-las utilizando algoritmos padrão, catalogados na literatura e específicos para essas estruturas de dados. Entre outras necessidades, essa abordagem é utilizada para pré-processar os símbolos presentes na estrutura (*Go* #3) e criar *Estruturas Auxiliares de Indexação* (ver Seção 4.7), ou mesmo para coletar nós ao implementar uma funcionalidade como a `Find References`. Os algoritmos utilizados já são conhecido na literatura. Exemplos de algoritmos dessa categoria são, a *Busca em Profundidade* ou a *Travessia de Árvore In-Order* (CORMEN et al., 2022).

Adhoc (10 servidores): Algoritmos mais simples ou que não seguem uma estratégia padrão, descrita na literatura. Um exemplo seria a funcionalidade `Completion` implementada no servidor *XML* (#7), sua implementação simplesmente navega por todos os nós ancestrais (*parent nodes*), os coletando para então sugerir itens que podem ser utilizados para completar um símbolo. Outra utilização da navegação de baixo para cima é a identificação do escopo onde um símbolo é declarado (ex., quando o símbolo é um atributo dentro da árvore de um nó ancestral do tipo *Classe*). Finalmente, implementações de travessia customizadas são também utilizadas para determinar se a posição do cursor está contida em alguma estrutura semântica, como o lado direito de uma instrução de atribuição (*Flux* #5), que necessita a navegação entre os nós irmãos ou primos do nó alvo.

Visitor (10 servidores): De maneira estratégica (ex., utilizando funções lambda ou o padrão *Strategy*), são definidos filtros que indicam características dos símbolos desejados. Um orquestrador atravessa a árvore para coletar ou notificar o código que o invoca, sobre quais nós da árvore que correspondem ao desejado. Esse é um padrão de design estabelecido em Gamma et al. (1995), chamado *Visitor*. Como resultado, as implementações das funcionalidades se tornam mais simples dado a dispensa em controlar a ordem de travessia dos nós. Esse é um padrão comumente disponibilizado por compiladores ou *Bibliotecas de Suporte à Linguagem*, um exemplo é a classe *TreePathScanner* (ORACLE, 2022) fornecida pelo projeto *javac* e utilizada pelo servidor *Java* (#23). Outro exemplo de

Tabela 10 – Relação de estratégias de travessia utilizadas pelos servidores

	Adhoc	**Padrão**	**Visitor**	**Terceiro**	**
Adhoc		3	3	1	
Padrão	3	3	1	2	
Visitor	3	1		4	
Terceiro	1	2	4	2	
Nenhum					7
Padrão + Visitor + Terceiro					1
Padrão + Visitor + Adhoc					1
Padrão + Adhoc + Terceiro					2

A seção de transversais entre as quatro formas de travessia (Adhoc, Padrão, Visitor, Terceiro) demonstra as informações espelhadas com base na diagonal central.

servidor que utiliza uma implementação de *Visitor* para filtrar/aceitar nós contidos em uma árvore é a implementação da funcionalidade `Completion` no servidor *C/C++* (#9). Além de implementações baseadas em linguagens orientadas a objetos, outras variações utilizam funções *lambda* (ex., *rust-analyzer* utiliza o Hir (The Rust Foundation, 2022) para coletar resultados esperados). Algumas bibliotecas como a *JDT* e a *Clang/sema* vão ainda mais adiante e disponibilizam formas declarativas para especificar características de nós que serão aceitos e então as bibliotecas executam diretamente as funcionalidades (ex., `Completion`, `Hover` ou `Find References`), retornando o resultado a partir desses filtros. Finalmente, foram encontrados casos como o do servidor *R* (#2) que utiliza o XPath para a navegação da representação da árvore em DOM[5].

Terceiros (12 servidores): Nessa categoria se enquadram os servidores que utilizam bibliotecas de terceiro, responsáveis por orquestrar os visitadores ou mesmo por coletar as informações através de funções explicitas que requisitam, por exemplo, (i) todos os símbolos de um documento, ou (ii) todas as variáveis de um documento. Se enquadram aqui também os servidores que utilizam *Bibliotecas de suporte à edição* (ver Tabela 9) para executar as funcionalidades completamente.

Ao se avaliar os dados da tabela 10, pode-se verificar que a combinação mais comum de padrões é a *Visitor+Terceiro* (4 servidores), isso se deve ao fato de determinadas APIs de linguagem como o *javac*, fornecerem estratégias para travessia das árvores resultantes. O *Visitor* em combinação com o *Adhoc* também é encontrado com maior frequência (3 servidores), demonstrando que mesmo em combinação com estratégias manuais e desestruturadas de navegação e sem auxílio de bibliotecas de terceiros, o *Visitor* é uma metodologia viável de travessia.

Em uma análise separada das estratégias, as navegações por algoritmos *Padrões* e através do código de *Terceiros* são as únicas que são utilizadas sozinhas em alguns servidores (sem nenhuma outra estratégia). A utilização de somente navegação através

[5] <https://www.w3.org/TR/1999/REC-xpath-19991116/>

de código de terceiros demonstra um alto potencial de reúso das bibliotecas externas, ao mesmo tempo a utilização de estratégias padrões de árvore/grafo sugerem que há potencial de implementar os algoritmos de navegação utilizando esses algoritmos e depois reutilizá-los na implementação das funcionalidades.

A implementação *Adhoc* nunca é encontrada sozinha, um possível motivo é o fato de ser muito específica para resolver detalhes de uma única funcionalidade que necessita identificar o contexto onde os nós se encontram. Logo são um recurso útil para validações ou prover funcionalidades mais refinadas, porém dificilmente serão reutilizados para buscar itens explorando toda a árvore.

A combinação de mais que duas estratégias também foi identificada (4 servidores), porém em menor escala, enquanto a utilização *Nenhuma* estratégia de travessia (7 servidores) é comum em servidores que não implementam nenhuma funcionalidade ou que delegam sua implementação para *Bibliotecas de Suporte à Linguagem*.

Concluindo, para implementar a travessia, recomenda-se que os engenheiros primeiro procurem por bibliotecas utilizadas em produção e que forneçam um suporte sofisticado para representar e atravessar as instâncias do código-fonte. Se nenhuma biblioteca está disponível para a linguagem a ser suportada, então deve-se implementar a travessia através de estratégias que abstraiam a navegação, como o padrão *Visitor*, pois são úteis para criar componentes altamente reutilizáveis e que serão base para a implementação da camada de *Suporte à Edição*. Caso nenhuma opção de biblioteca para compilar e criar a representação abstrata esteja disponível, uma ferramenta de criação de *parsers* a partir de gramáticas de linguagem (ex., *Xtext* ou *Antlr*) pode ser utilizada. Somente, e então somente, se ainda não for possível, ou houverem boas razões para não utilizar uma biblioteca de terceiro ou criar componentes reutilizáveis para travessia, os desenvolvedores desses servidores devem escrever algoritmos *Adhoc* para implementar as funcionalidades.

4.7 Aspecto 6: Otimização de performance

Se for levado em consideração um suporte à edição eficiente e uma boa usabilidade, o tempo de resposta do sistema é um requisito crítico para a aceitação por parte dos desenvolvedores. Por esse motivo, espera-se observar estratégias de otimização (ex., implementações de cache) sendo utilizadas para melhorar a performance de execução das funcionalidades. Um exemplo de otimização seria, evitar a reinterpretação do código-fonte na execução de cada chamada de funcionalidade para melhorar a performance geral do servidor. Ao investigar a amostra, foram identificadas três estratégias de otimização comuns:

Cache da AST: O processo de *parsing* pode ser custoso, principalmente se realizado a cada requisição. Para evitar que essa atividade ocorra constantemente, uma

solução adotada comumente (23 servidores) é o armazenamento em memória, através de uma relação chave-valor entre o identificador do documento e a representação gerada. O armazenamento combina a URI do documento (como chave) com a estrutura de dados resultante do processo de compilação/*parsing*, por exemplo o nó raiz de uma AST (como valor).

A estrutura de dados pode ser criada quando o cliente notifica que o documento foi aberto (como no servidor *XML* #7) ou mesmo em uma fase de inicialização do servidor, onde todos os arquivos do espaço de trabalho são compilados e armazenados em cache. Quando essa indexação dos documentos compilados ocorre somente ao abrir o documento atual, uma preocupação adicional dos engenheiros é em identificar e rastrear os documentos adicionais do qual o primeiro depende (ex., importações na linguagem *Java*). Como usual em qualquer estrutura de cache, é necessária a implementação de mecanismos para invalidar, ou mesmo atualizar, a representação abstrata quando mudanças ocorrerem no documento, assunto da Seção 4.8.

Estrutura Auxiliar de Indexação: Uma estratégia utilizada por 13 servidores para acessar a representação da instância do código-fonte, porém sem operar diretamente sobre essa representação, é prover uma *Estrutura Auxiliar de Indexação* que contêm nós relevantes do arquivo fonte. A estratégia é baseada (i) na travessia da *Representação Abstrata*, (ii) coleta dos nós relevantes, e (iii) armazenamento desses nós em uma estrutura de dados indexada. Uma forma concreta de se representar tal estrutura indexada, é a implementação de uma *Tabela de Símbolos* (ver 2.3). Como a estrutura indexada armazena informações sobre os símbolos contidos em um documento, isso pode simplificar a implementação de funcionalidades LSP. Os engenheiros então, não necessitam atravessar a representação a todo momento e diferenciar entre os diferentes tipos de nós para cada funcionalidade implementada. Também auxilia com a performance, pois as informações coletadas uma vez podem ser reutilizadas em diferentes chamadas de funcionalidades, apenas explorando uma estrutura mais simples (ex., Dicionário ou Lista). Assim como nas representações abstratas de instâncias, essa estrutura indexada também necessita ser reprocessada toda vez que documento é modificado, estratégia utilizada no servidor *Cobol* #15.

Escaneamento do Escopo: Além das representações abstratas, para auxiliar na implementação das funcionalidades do LSP, 22 servidores também operam diretamente no código-fonte. Dadas as limitações em operar o código diretamente, como a falta de contexto para os símbolos, essas operações são utilizadas somente em casos específicos, porém evitando a carga necessária de construir estruturas de dados adicionais. Operações simples de manipulação de texto, como navegação caractere a caractere, podem ser utilizadas para escanear o código-fonte e coletar informações simples (ex., em qual posição o símbolo atual inicia e termina). Essas informações podem também ser obtidas através

da utilização de Expressões Regulares, aplicadas para ambos, o código-fonte completo ou somente no conteúdo de uma linha, como implementado pelo servidor *Elixir* (#16). Uma observação para os engenheiros é que tais informações avaliadas via texto podem usualmente ser coletada dos próprios atributos dos nós da AST, sem a necessidade da manipulação direta do conteúdo do código-fonte.

Outra forma de utilização do escaneamento é a extração de informações diretamente do código-fonte ao relacionar uma posição específica do código-fonte com outras posições. Por exemplo, ao verificar se um símbolo selecionado esta contido entre parênteses. Em posse de tais informações, os engenheiros podem determinar se o contexto do símbolo é uma seção de declaração de parâmetros de uma função (*Xtext* #13). Outra aplicação desse tipo de lógica é na identificação de algum carácter especial precedendo o símbolo atual (*Assembler* #6) ou para identificar se o cursor está dentro de um bloco de comentários (*R* #2).

Essas otimizações focam principalmente em reduzir o tempo gasto na construção de instâncias intermediárias para representar o código-fonte toda vez que uma funcionalidade é requisitada, bem como, no carregamento rápido e indexado dessas representações.

4.8 Aspecto 7: Lidando com mudanças nos documentos

Ao prover suporte à edição para linguagens, não se pode esperar que os documentos sejam recursos estáticos, pois eles estarão em constante mudança. Os usuários modificarão esses arquivos, e as representações necessárias se tornarão obsoletas em um curto intervalo de tempo, no caso do documento em edição. Isso se aplica também para dependências ou arquivos dependentes, no caso de refatorações (ex., ao renomear uma função contida em uma classe). O protocolo LSP já define abordagens para o cliente notificar o servidor que mudanças ocorreram nos documentos ou que eles foram salvos (MICROSOFT, 2022b).

Funcionalidades como a `Diagnostics` são sensíveis à mudanças na instância do código-fonte. Qualquer otimização de performance, como armazenamento em memória da representação abstrata ou mesmo estruturas indexadas devem ser atualizadas quando elas não refletem mais o estado atual do documento. A especificação do LSP estabelece duas requisições, `textDocument/didChange` e `textDocument/didSave`, que são enviadas do cliente para o servidor para notificar que o estado dos recursos em edição foram atualizados. Para o caso do `didChange`, o conjunto de mudanças que aconteceram no recurso são enviadas ao servidor, que pode lidar com elas *Incrementalmente* ou *Completamente*, deixando a representação do recurso no servidor no mesmo estado que do lado do cliente. O `didSave` irá notificar quando o documento foi salvo e pode conter opcionalmente o conteúdo do arquivo ao salvá-lo.

Uma sincronização incremental significa que o servidor receberá a lista de modifi-

cações que foram realizadas no documento, podendo ser uma lista de linhas modificadas ou de operações de texto realizadas (ex., intervalos de texto alterados ou excluídos) e poderá aplicá-las e reavaliar o documento incrementalmente. A atualização completa, como o nome indica, sinaliza ao servidor que o conteúdo do arquivo deve ser totalmente recarregado de seu local de armazenamento ou mesmo enviar para o mesmo no corpo da requisição o novo conteúdo completo do documento, que deve então ser reprocessado.

Tabela 11 – Formas de gerenciar mudanças nos servidores da amostra

	Strategies	
	reprocess	invalidate
textDocument/didChange	13	7
textDocument/didSave	5	2

Dos servidores avaliados, o caso mais comum foi a sincronização do servidor de forma incremental, 19 servidores, enquanto 9 servidores realizaram a sincronização completa do conteúdo. Dois servidores não sincronizam o conteúdo dos documentos, *SonarLint* (#24) que somente executa análise estática dos documentos e não implementa nenhuma outra funcionalidade e o *Turtle* (#8), que recarrega o conteúdo dos documentos a partir do disco.

As estratégias para sincronizar a representação do conteúdo são (i) *reprocessar* o arquivo, interpretá-lo novamente e recriar a representação abstrata; (ii) *invalidar* o arquivo, sinalizando-o como obsoleto ou limpando a representação em memória, o arquivo então será reprocessado quando a próxima funcionalidade que o utilizar for requisitada. De acordo com a Tabela 11, para os servidores avaliados: 13 servidores reprocessaram quando o `didChange` acontece, 7 invalidam quando o `didChange` acontece, 5 servidores reprocessam quando o `didSave` acontece e 2 servidores invalidam o arquivo quando o `didSave` acontece. Uma possível explicação para a popularidade do reprocessamento das representações instantaneamente após as mudanças nos arquivos, é devido a necessidade de se ter sempre representações atualizadas do código, pois ao criar uma nova função ou variável, mesmo antes de salvar o documento, as funcionalidades de edição devem estar disponíveis, para completar o código com esse novo símbolo.

Existem casos em que a representação abstrata não precisa ser atualizada, como quando uma *Biblioteca de Suporte à Linguagens* é reutilizada ou mesmo quando o servidor sempre reprocessa o conteúdo em cada requisição, ou até mesmo quando não há representação para ser utilizada como no servidor *SonarLint* (#24). A funcionalidade `Diagnostics` é sensível as mudanças e pode ser implementada quando essas requisições acontecem, 17 servidores utilizam o utilizam o `didChange` para prover diagnósticos para o cliente e 9 servidores quando o `didSave` acontece.

5 Discussão e Lições Aprendidas

Após a análise geral dos resultados, algumas observações gerais, bem como destaques para a implementação dos servidores emergiram e serão apresentados no presente capítulo.

5.1 Observações gerais

Relembrando que, prover um bom suporte à edição requer a implementação de uma variedade de funcionalidades de edição. Dada essa necessidade, a etapa do estudo sobre quais as funcionalidades de edição mais implementadas, em conjunto com suas relações com os paradigmas das linguagens, pode guiar os desenvolvedores LSP ao priorizar a implementação dessas funcionalidades, de forma que cubram a maior variedade de necessidades do projeto e explorem de forma abrangente o espaço de design do servidor, para então criar uma arquitetura sólida.

Um ponto interessante é que nem sempre se faz necessária a implementação de um processamento de linguagem pesado ou utilizando-se de técnicas já catalogadas na literatura. Por exemplo, observa-se que a utilização de outras representações de instâncias de linguagem, além da AST, pode ser suficiente de acordo com as funcionalidades implementadas.

A travessia da árvore é implementada repetidamente para diferentes propósitos, tendo um alto potencial de reúso. Também, *estruturas auxiliares indexadas*, contendo símbolos de interesse coletados da árvore, podem evitar repetidas travessias da AST em cada requisição de funcionalidade. Criar essas estruturas reutilizáveis é uma complexidade a mais para ser tratada durante o desenvolvimento do servidor, porém reduz a complexidade da implementação das funcionalidades.

A arquitetura em camadas identificada pode facilmente suportar tais mecanismos reutilizáveis e flexibilizar a conexão de diferentes implementações. Prover um suporte à edição mais sofisticado, por exemplo, suportando múltiplas linguagens ou análises mais sofisticadas, resultará em uma transição de um projeto com classes únicas por funcionalidade, para, como observado na análise deste trabalho, múltiplas classes por funcionalidade. Um ponto a ser levado em consideração é a separação entre a implementação do servidor e do suporte à edição em diferentes projetos, assim o suporte à edição fica encapsulado em uma *Biblioteca de Suporte à Edição*, que pode ser reutilizada para outras necessidades, além do LSP.

No geral, bibliotecas podem auxiliar na implementação do suporte à edição, mas não podem realizar todo o trabalho, gerando inclusive preocupações adicionais. Por exemplo,

identificar formas de encapsular o código reutilizado dessas bibliotecas externas, evitando assim poluir as *interfaces* do projeto.

Indo mais adiante na discussão sobre reúso, um suporte à edição gerado automaticamente pode ser uma forma de lidar com as necessidades padrão de um projeto (ex., funcionalidades mais comuns ou genéricas). A maioria dos servidores para DSLs, em oposição aos servidores para GPLs, utilizam o ANTLR (4 servidores) para a sua implementação, mas somente para a geração de *parsers*. Uma vez com os *parsers* em mão, as funcionalidades de edição são escritas manualmente, similar aos outros servidores. Porém, as implementações de LSP para linguagens baseadas no Xtext, tem uma implementação mais refinada e automatizada do que as escritas a mão. Essas implementações não são específicas de uma linguagem, mas de qualquer uma que possa ser expressa se utilizando de uma gramática Xtext. Por exemplo, em uma implementação da funcionalidade `Completion`, possíveis símbolos são identificados a partir da gramática definida pelo Xtext. Ao final, a implementação do suporte à edição é o mesma que o Xtext gera para um *plugin* de edição para a IDE Eclipse[1].

Todas as linguagens suportadas pelos servidores da amostra são baseadas em *parsers*. Um paradigma complementar a esse seria o de *Edição Projecional*, também conhecido por Orientado a Sintaxe ou de Edição Estruturada (BERGER et al., 2016; VÖLTER et al., 2014). As linguagens expressas com esse paradigma não utilizam *tokens* textuais para representar suas instância de programação, ao invés, movimentações em modelos gráficos realizadas pelos usuários alteram a AST subjacente, sem que nenhum processo de *parsing* ocorra. Os principais benefícios de tal tipo de edição são basicamente, (i) elas podem gerar composições ilimitadas de linguagens, e (ii) também é possível gerar notações flexíveis (textuais e visuais) para a linguagem. Enquanto não foi observado nenhum suporte específico para esse tipo de paradigma, o fato que servidores LSP vem sendo desenvolvidos para linguagens de modelagem gráfica, como a Eclipse GLPS[2], demonstra que o protocolo pode também ser utilizado para linguagens baseadas em Edição Projecional, como as implementadas utilizando o *Jetbrains Meta Programming System* (MPS)[3]. A principal preocupação para essas linguagens será em formas de identificar a localização dos elementos sendo editados, que não pode ser baseada em números de linha e coluna (como o definido pelo LSP), mas requer outro mecanismo de localização (ex., node ID).

5.2 Destaques identificados durante a análise

De acordo com os desafios levantados, serão apresentadas os principais destaques relacionados à arquitetura/implementação, a partir dos resultados apresentados

[1] <https://www.eclipse.org/Xtext/documentation/340_lsp_support.html>
[2] <https://www.eclipse.org/glsp/>
[3] <https://www.jetbrains.com/pt-br/mps/>

anteriormente.

- **Implementação das funcionalidades mais populares:** Como já reportado nesse trabalho, as funcionalidades mais populares dependem de um conjunto diverso de necessidades, grande o bastante para explorar o espaço de design do sistema. Desde a compreensão do código, através da coleta de dados sobre símbolos simples, até a exploração total do *workspace* (compreendendo o documento atual e suas dependências). Dessa forma, implementando somente essas funcionalidades, o design do projeto irá abranger um escopo de implementação que partirá da gestão dos documentos do espaço de trabalho, até a compilação do código, travessia, e qualquer necessidade de otimização. Uma vez implementadas essas funcionalidades, diversas oportunidades de reúso se revelarão e o arquétipo do restante do projeto será sólido o bastante para continuar a sua evolução. Dado as interdependências entre as funcionalidades, é oportuno iniciar a implementação das funcionalidades a partir do conjunto de funcionalidades: `Find References`, `Diagnostics` e `Goto Definition`.

- **Avaliação de bibliotecas disponíveis:** Projetos de suporte a edição com foco em linguagens populares e com grandes comunidades, tendem a já dispor de um maior número de bibliotecas. Sejam essas bibliotecas, para prover o suporte à edição completo para essas linguagens, ou para auxiliar em atividades pontuais (ex., gestão de documentação e análise estática de código). Mesmo em projetos menores, se demonstra mandatório um momento pré-desenvolvimento para avaliar as opções de compiladores, se possível até de *Bibliotecas de Suporte à Edição* já disponíveis e outros tipos de bibliotecas necessários.

- **Recursos disponibilizados pelo compiladores/*parsers* existentes para a linguagem:** O compilador/parser da linguagem suportada, com grande frequência é a biblioteca central para implementar o suporte à edição para servidores de linguagem. Os desenvolvedores se beneficiam de uma API simples fornecida por essa categoria de biblioteca, também deve-se levar em consideração os recursos disponibilizados pelos compiladores para navegar sobre as representações abstratas geradas. Bibliotecas que disponham de abstrações como o padrão *Visitor*, para navegar e coletar os nós da árvore, diminuirão a complexidade envolvida na implementação dessa tarefa, que será obrigatória para a execução das funcionalidades.

- **SDKs para auxiliar a implementação da infraestrutura do servidor:** As SDKs listadas na página da especificação LSP servem também como ferramentas para acelerar o desenvolvimento do projeto. Além de prover abstrações que ajudam os desenvolvedores na identificação de quais são os esquemas de mensagens necessários para implementar as funcionalidades. Esses SDKs são recursos que beneficiam os

desenvolvedores na criação de seus projetos, porém, é necessário que se verifique se a SDK selecionada suporta em tempo aceitável novas versões da especificação.

- **Encapsulamento das camadas do servidor:** Dadas as três camadas para a implementação do servidor identificadas neste trabalho, um encapsulamento adequado das lógicas de cada uma pode auxiliar na manutenção e favorecer o reúso entre essas camadas. Dado que as camadas de *Suporte à Linguagem*, bem como de *Infraestrutura do Servidor*, podem reaproveitar códigos de bibliotecas externas (ex., compiladores/*parsers*) ou de SDKs LSP, essas camadas podem servir como fachadas entre as interfaces do projeto e das bibliotecas, evitando assim um acoplamento forte com as mesmas.

- **Implementação do Suporte à Edição como uma biblioteca reutilizável:** Como citado na Seção 4.3.2, existem *Bibliotecas de Suporte à Edição* que implementam as funcionalidades de edição e podem ser reutilizadas pelos servidores LSP. Esses servidores somente implementam o protocolo necessário e então delegam a execução para essas bibliotecas. Essas bibliotecas podem ser reutilizadas em outros projetos, além das implementações de servidores LSP. Encapsular o *Suporte à Edição* em um projeto que possa ser utilizado como *Biblioteca de Suporte*, permite que outros projetos, além do servidor sendo criado, se beneficiem dessa implementação, permitindo assim que uma maior comunidade possa dar suporte a mesma.

- **Granularidade de implementação:** *Uma classe/arquivo por funcionalidade* é a granularidade de implementação mais comum, fornecendo um mínimo de encapsulamento e separação de código entre as funcionalidades. Caso hajam necessidades mais complexas, como suportar mais linguagens, ou as implementações de determinadas funcionalidades se demonstrem demasiadamente complexas, então uma granularidade mais refinada deve ser considerada.

- **API de representação e travessia de instâncias do código-fonte:** Representar e atravessar as instâncias de código-fonte é o maior esforço para a implementação das funcionalidades. Uma implementação rica de travessia (ex., através do padrão *Visitor*) pode reduzir a complexidade da implementação das funcionalidades. Outra API rica que demonstre as nuances da árvore abstrata é também um recurso fundamental para os desenvolvedores. Exemplos de requisitos para essa API são: (i) retornar todos os irmãos de um nó, ou (ii) retornar os documentos dos quais o documento atual depende (ex., importações no caso da linguagem Java). Em um nível mais abstrato de implementação, funções como, retornar o nó da função que declara o parâmetro atual e mesmo seus parâmetros irmãos. Todos esses são pontos a serem levados em consideração, criando assim uma API específica da linguagem a ser suportada.

- **Implementação de uma Estrutura Auxiliar Indexada:** Navegar constantemente a representação do código para cada funcionalidade significa um grande esforço de implementação e um número considerável de cláusulas de decisão (para testar os tipos dos nós que estão sendo atravessados). Coletar atalhos para cada nó durante a primeira travessia da árvore em uma estrutura indexada pode diminuir o esforço para implementar funcionalidades como `Document Symbols`. Tal estrutura evita que a travessia ocorra diversas vezes, bem como resulta em uma estrutura de dados mais simples de se consumir do que representações em árvores/grafos (ex., um dicionário).

- **Informações coletadas pelo escaneamento do código-fonte podem ser encontradas na representação abstrata:** Os desenvolvedores devem ter um conhecimento acurado da API do compilador/*parser* escolhido, bem como dos esquemas de dados utilizados para representar os nós. Informações que os desenvolvedores coletam diretamente do código fonte, podem estar presentes nos nós da representação abstrata. Exemplos, posição do nó no código-fonte ou qual o símbolo anterior ao nó selecionado.

- ***Publish/Subscribe* entre as camadas:** O estado dos documentos é uma informação sensível para a orquestração do estado geral do servidor, bem como para a execução de determinadas funcionalidades. A implementação de padrões como o *Observer* (GAMMA et al., 1995) pode ser utilizada para rastrear mudanças nos documentos e notificar por exemplo, (i) a camada de *Suporte à Edição* que os diagnósticos devem ser coletados e enviados para o cliente, ou (ii) a camada de *Suporte à Linguagem* que a representação abstrata do documento está obsoleta e qualquer cache deve ser invalidado ou a representação deve ser recriada em memória. Esses são exemplos de como as informações podem ser sincronizadas entre as camadas, quando ocorrem operações assíncronas, garantindo um único fluxo de dados entre essas camadas.

Uma vez com essas destaques listados, espera-se que possam servir como referência para engenheiros que estejam criando ou modernizando servidores LSP. Esses pontos, também podem auxiliar desenvolvedores que desejam criar *plugins* e ferramentas de suporte à edição que não dependam do protocolo LSP.

6 Ameaças à Validade

Validade Externa: Primeiro, o foco no LSP ao investigar o suporte à edição pode enviesar os resultados desnecessariamente dado as tecnicalidades definidas pelo protocolo, limitando a generalização ao suporte à edição em geral. De qualquer forma, o LSP é um protocolo amplamente utilizado para prover suporte à edição independente da ferramenta e focando principalmente na comunicação entre o cliente e servidor, o que torna os resultados do trabalho mais gerais. Além disso o protocolo de comunicação é somente uma parte menor da implementação dos servidores. Segundo, a seleção de servidores LSP poderia enviesar os achados (aspectos e práticas). Para mitigar esse desafio, foi seguido um processo de seleção em quatro etapas para obter uma amostra adequada. Foi também coberta uma boa variedade de linguagens, a amostra de servidores prove suporte para 31 linguagens diferentes. Essa amostra de servidores também depende de 14 linguagens de implementação diferentes, reduzindo a possibilidade de viés nos achados. Terceiro, os resultados poderiam ser enviesados entre linguagens de modelagem específicas de domínio ou linguagens de programação. De qualquer forma, a amostra contêm 11 da primeira categoria e 19 da segunda, reduzindo o viés.

Validade Interna: Primeiro, durante a análise, pode-se ter compreendido mal os detalhes internos de alguns servidores. De qualquer forma, a estratégia de seleção garante que também fossem investigados os servidores baseados em Java, acreditando na experiência dos autores com ferramentas de engenharia de linguagem baseadas em Java. Segundo, foi primeiro observada a arquitetura e então as implementações, especificamente, os pontos de entrada foram as 7 funcionalidades implementadas mais frequentemente. Essa seleção poderia não compreender práticas importantes observadas somente em outras funcionalidades. De qualquer forma, o objetivo do trabalho de dar suporte para outros desenvolvedores ao implementar seus próprios servidores ou as funcionalidades 'populares', justifica esse escopo e vieses em potencial. Terceiro, foi utilizado o método bem estabelecido de Análise Temática. Depois de coletar os dados de cada servidor, os colaboradores discutiram extensivamente os códigos, rótulos e preocupações refinadas. Sendo dois desses colaboradores especialistas em linguagens de programação e DLSs. Quarto, dado que a extração dos segmentos relevantes, bem como a codificação dos resultados for realizada exclusivamente pelo primeiro autor, a revisão de cada segmento, em cada etapa, foi necessária por parte dos colaboradores (especialistas). O objetivo dessa revisão foi evitar o enviesamento de uma análise subjetiva.

Conclusão sobre a validade: Mesmo que as práticas de implementação identificadas e discutidas nesse trabalho forneçam uma visão detalhada para quem deseja implementar o suporte à edição, elas podem ser consideradas limitadas somente aos seus

achados e populações semelhantes. Ainda, o trabalho constrói uma fundação sólida para futuras avaliações sobre o menor esforço necessário para a implementação, bem como sobre quais práticas além das identificadas podem existir para os desafios levantados.

7 Conclusão

O presente trabalho, investigou uma área de pesquisa pouco explorada, do Suporte à Edição para linguagens de software. Neste capítulo são apresentadas as principais contribuições desse trabalho inicial, bem como trabalhos futuros para complementar e avançar os achados apresentados.

7.1 Contribuições

O suporte à edição efetivo para linguagens de software é crucial para o sucesso do desenvolvimento de software. Foi apresentado nesse trabalho o primeiro conjunto de aspectos e praticas para a design e implementação de suporte à edição sistematicamente identificado (quantitativamente e qualitativamente) de uma amostra de 30 servidores LSP. Foram sintetizados sete aspectos centrais e apresentadas possíveis soluções para implementar o suporte à edição utilizando servidores LSP. Foi constatado que uma variedade de funcionalidades são necessárias para prover suporte à edição e os aspectos concretos das linguagens tem um papel menor no suporte básico a edição. Ainda, para um suporte à edição avançado, características da linguagem a ser suportada recebem maior relevância. Para o design de servidores LSP, foram identificadas práticas principais que serão estendidas em uma arquitetura de referência em trabalhos futuros. Em se tratando da implementação concreta dos servidores LSP e suporte à edição em geral, para várias tarefas de implementação existem bibliotecas maduras e que já são amplamente utilizadas. Adicionalmente, foram observadas múltiplas técnicas de otimização frequentemente utilizadas e que usualmente são acompanhadas do desafio de atualizar estruturas de cache. Também, para esse desafio foram coletadas formas de representar as instância de linguagens, atravessar essas representações e lidar com mudanças nos documentos que devem disparar atualizações nessas representações.

7.2 Trabalhos Futuros

Apesar de apresentar uma sólida base para explorar os desafios da implementação do Suporte à Edição para linguagens de software, trabalhos adicionais ainda são necessários para fornecer esse corpo de conhecimento, antes mesmo de propor uma arquitetura de referência ou padrões. Um trabalho inicial será a coleta de dados, também, sobre desafios de implementação de Suporte à Edição, porém dessa vez através da experiência dos desenvolvedores que mantém essas ferramentas.

Após identificar quais os desafios independentes das funcionalidades, análises aprofundadas de aspectos específicos para cada funcionalidade também se fazem necessárias. Dado as limitações de recursos para a execução deste trabalho, uma análise em profundidade sobre quais os componentes de códigos reaproveitados entre as funcionalidades (ex., funções e classes) ou entre as camadas de implementação é crucial para um entendimento mais refinado das práticas de implementação existentes.

Após a execução deste trabalho, foi identificada a necessidade de conhecer as APIs e recursos providos pelos compiladores/*parsers* para a implementação do suporte à edição, dado a sua importância e popularidade nos servidores da amostra. Catalogar e avaliar os recursos disponibilizados por *Bibliotecas de Suporte à Edição*, outras além das encontradas na análise deste trabalho, também se faz necessário.

Análise quantitativas das práticas de otimização de performance (ex., cache e estruturas auxiliares indexadas), além de outras avaliações e *benchmarks* envolvendo a implementação da arquitetura dos servidores se faz necessária. Este trabalho se atentou ao processo de catalogar essas práticas, a partir de agora uma análise criteriosa da eficiência das mesmas se faz necessária.

Referências

AGRAWAL, R.; SRIKANT, R. et al. Fast algorithms for mining association rules. In: *20th International Conference on Very Large Data Bases (VLDB)*. [S.l.: s.n.], 1994. v. 1215, p. 487–499. Citado na página 48.

AHO, A. V. et al. *Compilers: principles, techniques, & tools*. [S.l.]: Pearson, 2007. Citado 4 vezes nas páginas 14, 15, 24 e 25.

BARROS, D. et al. *Supplementary Material – Editing Support for Software Languages: Implementation Practices in Language Server Protocols 59*. 2022. Disponível em: <https://www.dropbox.com/sh/ncxbk8yz3b2m79k/AAAYVsyzgTB8qxyawrtIxsR9a?dl=0>. Citado na página 28.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. [S.l.]: Addison-Wesley Professional, 2003. Citado na página 55.

BASTEN, B. et al. Modular language implementation in rascal – experience report. *Science of Computer Programming*, v. 114, p. 7–19, 2015. ISSN 0167-6423. LDTA (Language Descriptions, Tools, and Applications) Tool Challenge. Citado na página 15.

BERGER, T. et al. Efficiency of projectional editing: A controlled experiment. In: *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. [S.l.: s.n.], 2016. Citado 2 vezes nas páginas 29 e 70.

BIALY, M. et al. Software engineering for model-based development by domain experts. In: GRIFFOR, E. (Ed.). *Handbook of System Safety and Security*. Boston: Syngress, 2017. p. 39–64. ISBN 978-0-12-803773-7. Citado na página 14.

BROWN, W. H. et al. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. [S.l.]: Wiley & Sons, 1998. Citado na página 55.

BÜNDER, H. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In: *MODELSWARD*. [S.l.: s.n.], 2019. p. 129–140. Citado 2 vezes nas páginas 14 e 19.

BÜNDER, H.; KUCHEN, H. Towards multi-editor support for domain-specific languages utilizing the language server protocol. In: SPRINGER. *International Conference on Model-Driven Engineering and Software Development*. [S.l.], 2019. p. 225–245. Citado na página 26.

BÜNDER, H.; KUCHEN, H. Towards multi-editor support for domain-specific languages utilizing the language server protocol. In: HAMMOUDI, S.; PIRES, L. F.; SELIĆ, B. (Ed.). *Model-Driven Engineering and Software Development*. Cham: Springer International Publishing, 2020. p. 225–245. ISBN 978-3-030-37873-8. Citado na página 26.

COMMUNITY, E. *XText*. 2022. Accessed: 2022-04-10. Disponível em: <https://www.eclipse.org/Xtext/>. Citado na página 24.

COOK, S. et al. Unified modeling language (uml) version 2.5. 1. *Object Management Group (OMG), Standard*, v. 12, 2017. Citado na página 14.

COOPER, K. D.; TORCZON, L. *Engineering a compiler*. [S.l.]: Elsevier, 2011. Citado 2 vezes nas páginas 23 e 24.

CORMEN, T. H. et al. *Introduction to algorithms*. [S.l.: s.n.], 2022. Citado 2 vezes nas páginas 61 e 63.

COULON, F. et al. Modular and distributed ide. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. [S.l.: s.n.], 2020. p. 270–282. Citado na página 27.

CRUZES, D. S.; DYBA, T. Recommended steps for thematic synthesis in software engineering. In: *International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2011. p. 275–284. Citado 2 vezes nas páginas 31 e 32.

DRAGULE, S. et al. A survey on the design space of end-user-oriented languages for specifying robotic missions. *Software and Systems Modeling*, Springer Science and Business Media LLC, fev. 2021. Citado 2 vezes nas páginas 14 e 29.

EASTERBROOK, S. et al. Selecting empirical methods for software engineering research. In: *Guide to advanced empirical software engineering*. [S.l.]: Springer, 2008. p. 285–311. Citado na página 32.

Eclipse Foundation. *Eclipse Buildship: Gradle integration for the Eclipse IDE*. 2022. Accessed: 2022-04-10. Disponível em: <https://github.com/eclipse/buildship>. Citado na página 53.

Eclipse Foundation. *Eclipse Java development tools (JDT)*. 2022. Accessed: 2022-04-10. Disponível em: <https://www.eclipse.org/jdt/>. Citado na página 54.

Eclipse Foundation. *Eclipse LSP4J*. 2022. Accessed: 2022-04-10. Disponível em: <https://github.com/eclipse/lsp4j>. Citado na página 54.

Eclipse Foundation. *M2Eclipse*. 2022. Accessed: 2022-04-10. Disponível em: <https://www.eclipse.org/m2e/>. Citado na página 53.

Elixir Language Server Protocol. *ElixirSense*. 2022. Accessed: 2022-04-10. Disponível em: <https://github.com/elixir-lsp/elixir_sense>. Citado na página 52.

ERDWEG, S. et al. The state of the art in language workbenches. In: *SLE*. [S.l.: s.n.], 2013. Citado 2 vezes nas páginas 14 e 25.

Ericsson AB. *Dialyzer User's Guide*. 2022. Accessed: 2022-04-10. Disponível em: <https://www.erlang.org/doc/man/dialyzer.html>. Citado na página 53.

FEKETE, A.; PORKOLÁB, Z. A comprehensive review on software comprehension models. *Annales Mathematicae et Informaticae*, Annales Mathematicae et Informaticae - AMI, v. 51, p. 103–111, 2020. Citado na página 18.

FOWLER, M. *Domain-specific Languages*. [S.l.]: Pearson Education, 2010. Citado na página 14.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. [S.l.]: Pearson Deutschland GmbH, 1995. Citado 5 vezes nas páginas 54, 55, 57, 63 e 73.

GLASER, B. *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory*. Sociology Press, 1978. (Advances in the methodology of grounded theory). ISBN 9781884156014. Disponível em: <https://books.google.at/books?id=73-2AAAAIAAJ>. Citado na página 30.

GRISS, M. L. Software reuse: From library to factory. *IBM systems journal*, IBM, v. 32, n. 4, p. 548–566, 1993. Citado na página 50.

GUNASINGHE, N.; MARCUS, N. *Language Server Protocol and Implementation*. [S.l.]: Apress, 2022. Citado 4 vezes nas páginas 15, 23, 26 e 39.

HALTER, D. *Jedi - an awesome autocompletion, static analysis and refactoring library for Python*. 2022. Accessed: 2022-04-10. Disponível em: <https://jedi.readthedocs.io/en/latest/>. Citado 2 vezes nas páginas 52 e 54.

HEBIG, R. et al. Model transformation languages under a magnifying glass – a controlled experiment with xtend, atl, and qvt. In: *26th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. [S.l.: s.n.], 2018. Citado na página 29.

LÄMMEL, R. *Software Languages*. [S.l.]: Springer, 2018. Citado 3 vezes nas páginas 14, 18 e 23.

LÄMMEL, R.; VISSER, E.; VISSER, J. Strategic programming meets adaptive programming. In: *Proceedings of the 2nd international conference on Aspect-oriented software development*. [S.l.: s.n.], 2003. p. 168–177. Citado na página 63.

LEE, P. M. *Upper Critical Values for Spearman's Rank Correlation Coefficient R_s*. 2005. Disponível em: <https://www.york.ac.uk/depts/maths/tables/>. Citado na página 47.

LILLACK, M.; BERGER, T.; HEBIG, R. Experiences from reengineering and modularizing a legacy software generator with a projectional language workbench. In: *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC)*. [S.l.: s.n.], 2016. Citado na página 29.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 37, n. 4, p. 316–344, 2005. Citado na página 15.

MÉSZÁROS, M.; CSERÉP, M.; FEKETE, A. Delivering comprehension features into source code editors through lsp. In: IEEE. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. [S.l.], 2019. p. 1581–1586. Citado na página 18.

MICROSOFT. *An early-stage PHP parser designed for IDE usage scenarios*. 2022. Accessed: 2022-04-10. Disponível em: <https://github.com/microsoft/tolerant-php-parser>. Citado na página 51.

MICROSOFT. *Language Server Protocol*. 2022. Accessed: 2022-04-05. Disponível em: <https://microsoft.github.io/language-server-protocol/>. Citado 6 vezes nas páginas 15, 19, 20, 30, 44 e 67.

Microsoft. *VSCode Language Server - Node*. 2022. Accessed: 2022-04-10. Disponível em: <https://github.com/microsoft/vscode-languageserver-node>. Citado na página 54.

Mészáros, M.; Cserép, M.; Fekete, A. Delivering comprehension features into source code editors through lsp. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. [S.l.: s.n.], 2019. p. 1581–1586. Citado na página 26.

ORACLE. *Class TreePathScanner*. 2022. Accessed: 2022-04-10. Disponível em: <https://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/util/TreePathScanner.html>. Citado na página 63.

PARR, T. *Language Implementation Patterns: Create Your Own Domain-specific and General Programming Languages*. [S.l.]: Pragmatic Bookshelf, 2009. Citado 2 vezes nas páginas 14 e 15.

PARR, T. *ANTLR (ANother Tool for Language Recognition)*. 2022. Accessed: 2022-04-10. Disponível em: <https://www.antlr.org/>. Citado 2 vezes nas páginas 24 e 51.

PEARSON, K. Vii. mathematical contributions to the theory of evolution. iii. regression, heredity, and panmixia. *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character*, The Royal Society London, n. 187, p. 253–318, 1896. Citado na página 46.

PELDSZUS, S. *Security Compliance in Model-driven Development of Software Systems in Presence of Long-Term Evolution and Variants*. Springer, 2022. Disponível em: <https://www.semanticscholar.org/paper/0cd9a4d39917e3496e658caead1917f505f1d72a>. Citado na página 29.

PELDSZUS, S. et al. Incremental Co-Evolution of Java Programs based on Bidirectional Graph Transformation. In: STANSIFER, R.; KRALL, A. (Ed.). *Principles and Practices of Programming on The Java Platform (PPPJ)*. [S.l.]: ACM, 2015. p. 138–151. Citado na página 29.

PELDSZUS, S. et al. Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching. In: *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2016. p. 578–589. Citado na página 55.

PHPDOCUMENTOR. *ReflectionDocBlock*. 2022. Accessed: 2022-04-10. Disponível em: <https://github.com/phpDocumentor/ReflectionDocBlock>. Citado na página 52.

PUPO, A. L. S. et al. Guardiaml: Machine learning-assisted dynamic information flow control. In: IEEE. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2019. p. 624–628. Citado na página 26.

Python Code Quality Authority. *Pylint*. 2022. Accessed: 2022-04-10. Disponível em: <https://pypi.org/project/pylint/>. Citado na página 53.

QUEIROZ, R.; BERGER, T.; CZARNECKI, K. Geoscenario: An open dsl for autonomous driving scenario representation. In: *30th IEEE Intelligent Vehicles Symposium (IV)*. [S.l.: s.n.], 2019. Citado na página 29.

RDocumentation. *cor: Correlation, Variance and Covariance (Matrices)*. 2022. Accessed: 2022-04-10. Disponível em: <https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/cor>. Citado na página 46.

RODRIGUEZ-ECHEVERRIA, R. et al. Towards a language server protocol infrastructure for graphical modeling. In: *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. New York, NY, USA: Association for Computing Machinery, 2018. (MODELS '18), p. 370–380. ISBN 9781450349499. Citado 2 vezes nas páginas 19 e 25.

SCHAUSS, S. et al. A chrestomathy of dsl implementations. In: *10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. [S.l.: s.n.], 2017. Citado na página 29.

SEBESTA, R. W. *Concepts of programming languages*. [S.l.]: Pearson Education India, 2004. Citado na página 18.

SonarSource. *SonarLint Website*. 2022. Accessed: 2022-04-10. Disponível em: <https://www.sonarlint.org/>. Citado na página 46.

SOURCEGRAPH. *Langserver.org: A community-driven source of knowledge for Language Server Protocol implementations*. 2022. Accessed: 2022-04-05. Disponível em: <https://langserver.org/>. Citado 3 vezes nas páginas 15, 16 e 29.

SPINELLIS, D. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, v. 56, n. 1, p. 91–99, 2001. ISSN 0164-1212. Citado na página 15.

Stardog Union. *StarDog IDE*. 2022. Accessed: 2022-04-10. Disponível em: <https://www.stardog.com/studio/>. Citado na página 46.

STOLPE, D. et al. Language-independent development environment support for dynamic runtimes. In: *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. New York, NY, USA: Association for Computing Machinery, 2019. (DLS 2019), p. 80–90. ISBN 9781450369961. Citado na página 26.

STRÜBER, D.; PELDSZUS, S.; JÜRJENS, J. Taming Multi-Variability of Software Product Line Transformations. In: RUSSO, A.; SCHÜRR, A. (Ed.). *Fundamental Approaches to Software Engineering (FASE)*. Springer, 2018. (Lecture Notes in Computer Science, v. 10802), p. 337–355. Disponível em: <https://doi.org/10.1007/978-3-319-89363-1_19>. Citado na página 29.

The Rust Foundation. *Guide to Rustc Development: The HIR*. 2022. Accessed: 2022-04-10. Disponível em: <https://rustc-dev-guide.rust-lang.org/hir.html>. Citado na página 64.

VÖLTER, M. et al. *DSL Engineering - Designing, Implementing and Using Domain-specific Languages*. [S.l.]: M Volter / DSLBook.org, 2013. Citado na página 14.

VÖLTER, M. et al. Towards user-friendly projectional editors. In: *7th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. [S.l.: s.n.], 2014. Citado na página 70.

VÖLTER, M. et al. Efficient development of consistent projectional editors using grammar cells. In: *9th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. [S.l.: s.n.], 2016. Citado na página 29.

WASOWSKI, A.; BERGER, T. *Domain-specific Languages: Effective Modeling, Automation, and Reuse*. [s.n.], 2022. Disponível em: <http://dsl.design>. Citado 3 vezes nas páginas 14, 24 e 29.

XIA, X. et al. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, IEEE, v. 44, n. 10, p. 951–976, 2017. Citado na página 18.

Apêndices

APÊNDICE A – ARTIGO PUBLICADO NA XXV MODELS - INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEM

Editing Support for Software Languages: Implementation Practices in Language Server Protocols

Djonathan Barros
PPGComp, Western Paraná State University
Brazil

Sven Peldszus
Ruhr-University Bochum
Germany

Wesley K. G. Assunção
Johannes Kepler University
Austria

Thorsten Berger
Ruhr-University Bochum and Chalmers | University of Gothenburg
Germany and Sweden

ABSTRACT

Effectively using software languages, be it programming or domain-specific languages, requires effective editing support. Modern IDEs, modeling tools, and code editors typically provide sophisticated support to create, comprehend, or modify instances—programs or models—of particular languages. Unfortunately, building such editing support is challenging. While the engineering of languages is well understood and supported by modern model-driven techniques, there is a lack of engineering principles and best practices for realizing their editing support. Especially domain-specific languages—often created by smaller organizations or individual developers, sometimes even for single projects—would benefit from better methods and tools to create proper editing support.

We study practices for implementing editing support in 30 so-called language servers—implementations of the language server protocol (LSP). The latter is a recent de facto standard to realize editing support for languages, separated from the editing tools (e.g., IDEs or modeling tools), enhancing the reusability and quality of the editing support. Witnessing the LSP's popularity—a whopping 121 language servers are in existence today—we take this opportunity to analyze the implementations of 30 language servers, some of which support multiple languages. We identify concerns that developers need to take into account when developing editing support, and we synthesize implementation practices to address them, based on a systematic analysis of the servers' source code. We hope that our results shed light on an important technology for software language engineering, that facilitates language-oriented programming and systems development, including model-driven engineering.

CCS CONCEPTS

• **Software and its engineering → Language features**; **Formal language definitions**; **Context specific languages**.

KEYWORDS

Language engineering, code assistance, source code editor, implementation practices.

MODELS '22, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9466-6/22/10.
https://doi.org/10.1145/3550355.3552452

ACM Reference Format:
Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. 2022. Editing Support for Software Languages: Implementation Practices in Language Server Protocols. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3550355.3552452

1 INTRODUCTION

Software languages are paramount—not only to software engineering, but also to many other engineering disciplines that need to create models and automate tasks. Effectively using software languages—including programming languages, as well as domain-specific or general-purpose modeling languages [11]—requires effective editing support. Modern IDEs and modeling tools often come with sophisticated editing support to create, comprehend, and modify programs or models expressed in a certain language. Typical editing support features are code completion, syntax highlighting, error marking, formatting, and refactoring, among others.

Unfortunately, creating proper editing support for languages is difficult. While for mainstream software languages, the vendors of software engineering or modeling tools typically invest the necessary resources to realize editing support, domain-specific languages (DSLs) are often created by smaller organizations or individual developers. Sometimes, their use is limited to specific purposes or only a few projects. At the same time, such languages play a major role in increasing automation in software engineering [7, 64], and offering concise and semantically rich notations [9, 13]. As such, DSLs would especially benefit from better support to realize editing support—allowing their users focus on solving real problems, instead of wasting time with learning the exact use of individual DSLs.

Researchers and practitioners have worked intensively on methods and tools to build languages. Thanks to modern techniques, such as meta-modeling, automated mapping of abstract-to-concrete syntax, or model transformations, often integrated in modern and convenient language workbenches [21], the architectures of language infrastructures [30, 61] and the language engineering processes are well-understood [23, 64]. In addition to the technology on language engineering, there are implementation patterns for programming languages [39], for instance, on how to build abstract syntax trees (ASTs), on how to realize the tree walking and rewriting, and on how to realize tree pattern matching [5]. For DSLs, patterns for domain analysis, design, and implementation have been presented [33, 54]. Unfortunately, there is a lack of engineering principles and best practices for realizing the editing support of languages.

The language server protocol (LSP) [27, 35] is a recent initiative from 2016 to modularize editing support into the so-called language

servers. The LSP addresses the problem that language-specific editing support is currently deeply integrated into single IDEs or editors, preventing its reuse or extension for different tools. The LSP aims at enabling language engineers to make their languages, be it programming languages or DSLs, available to a wide range of editing tools while requiring minimal effort for adoption and reuse. The LSP describes a common API that can be implemented and reused by different clients (i.e., editors and IDEs) [35]. The tools connect to a language server, where individual editing support *features* can be requested for a program or model. The LSP calls these instances *documents*, which is the term we use in the remainder. Originally, LSP was proposed by Microsoft to provide core features for the IDE Visual Studio Code [35]. So far, a total of 23 features are defined in the protocol. It quickly became popular as a protocol used by different editors and languages [35]. As of now, 121 different language servers are listed on a community-driven curated list of LSPs [53].

We took this opportunity and investigated the design and realization of real language servers. Our goal was to identify implementation *concerns* related to the realization of language editing support. By analyzing a substantial sample of 30 LSP server implementations, we identified relevant concerns and the different realizations. We synthesized both into a set of practices on how to realize editing support that can be used by language engineers and researchers. Our working hypothesis was that engineering principles can be identified within existing language servers, paving the way to our long-term goal of establishing patterns and pattern catalogs for realizing language editing support. Among our sample, we analyzed the source code of the seven most popular LSP features. We followed a thematic synthesis method [12] by coding, analyzing, grouping, and reporting how editing features are implemented.

We hope to provide researchers with concerns related to the realization of editing support and insights on implementation practices for addressing them, to spark a discussion on best practices and eventually developing a theory and novel techniques on realizing effective editing support for languages. We hope that practitioners can use our concerns and discussed solutions as guidelines when implementing new servers or extending existing ones.

2 BACKGROUND

Language editing support assists developers when writing, comprehending, and changing documents (i.e., programs or models). Unfortunately, effective support needs to be language-specific, and realizing it can be costly and error-prone. Typical editing support, such as renaming, code navigation, or hover documentation, requires proper parsing and traversal of the source tree and knowing the specifics of each language [27]. Necessary engineering activities such as choosing a suitable parser, knowing the keywords, collecting runtime information, and formatting the source code properly, are not trivial. Even worse, users may prefer different editing tools, such as different IDEs and modeling tools, requiring editing support to be implemented multiple times for individual languages [50]. To this end, the API provided by the LSP, which can be implemented once and used by multiple editing tools, reduces redundancies and maintenance efforts for editing support implementations.

LSP Overview. The LSP [35] is a client-server protocol that enables the clients (i.e., the editing tools) to request editing support from *language servers*. The features that can be implemented are defined by the LSP specification [35]. Each LSP server can implement different editing support features depending on the respective language's characteristics or the desired editing features.

The actual editing support is encapsulated as individual methods representing a total of 23 editing support features currently specified by the LSP. A language server has, like its clients, access to the workspace that contains the documents for which editing support is required. Another abstraction, called *symbols* in the LSP, refers to any language element for which some editing support can be provided, such as identifiers, expressions, statements, or other structural concepts (e.g., classes, methods, constants).

LSP Features. Based on the feature descriptions presented in the LSP specification [35], we can classify the features into three different categories. Note that we list and explain concrete features and the category they belong to shortly, in Sec. 4.1, specifically Fig. 2.

Code assistance: The LSP offers seven features that assist developers when writing new code. Those features usually suggest computations of changes, such as renaming symbols, formatting the code, or suggesting code completions for a given prefix.

Code comprehension: The LSP offers 14 features to assist developers when exploring documents (e.g., source code). Those features usually do not change documents, but help developers by providing documentation and supporting the navigation through them.

Auxiliary: The LSP offers two features that allow LSP servers to provide additional capabilities not further specified by the protocol. One example is *Execute Command*, allowing the server to provide additional capabilities, such as refactorings.

LSP Workflow. To give a general understanding on the LSP implementations, we describe the typical internal workflow of how feature requests are executed, as proposed in the LSP [35]. Every feature request is independent of others and consists of four parts:

Action identification: When requesting a feature, the *Client* identifies the action that must be executed (e.g., *textDocument/completion*) and the execution's target document. To describe this target, the client sends a document identifier (*DocumentID*) and the *cursor position* (line and column) within the document.

Load Document: Based on the *DocumentID*, the server loads the target document. To do this, not only the client on which the document is being edited needs access to the documents but also the server.

Provide action: With access to the document, the server executes the feature, e.g., identifying what symbol needs to be completed and collecting the completion information. Some features, such as `Completion` or `Rename`, require the server to explore all documents.

Feature response: Finally, the server returns a response message. For instance, for the feature `Completion`, after collecting suggestions that can be inserted at the target position, the server responds with a set of *Completion Items*. Each item is a *Text Edit* operation that can be applied to perform the suggested completion on the document.

3 METHODOLOGY

Our study focuses on implementation practices for editing support in LSP servers. We aim to identify relevant concerns, which are aspects that should be considered when developing editing support,

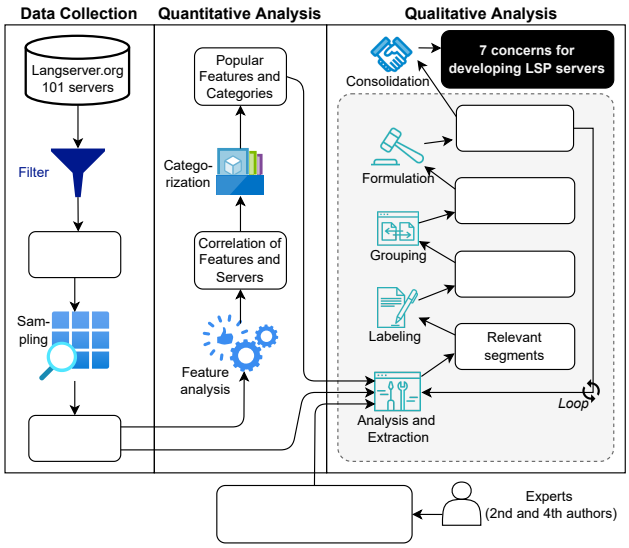

Figure 1: Methodology overview

and best practices addressing these concerns. Fig. 1 illustrates our methodology. First, a *quantitative analysis* illustrated in the center of Fig. 1 and described in Sec. 3.2, and second, a *qualitative analysis* illustrated on the right of Fig. 1 and described in Sec. 3.3. Further details are provided in our replication package [3].

3.1 Language Server Selection

As subjects for our study, we selected a sample of 30 LSP servers from a community-curated repository that collects known LSP servers [53]. While currently amounting to 121 LSPs (May 2022), at the time of the sample selection it had 101 LSPs (March 2021). We selected our sample as follows:

No Source Code. First, we filtered out all LSP servers for which we could not find their source code. This affected three servers.

Not Actively Developed. We removed 11 LSP servers listed as archived or deprecated, taking into account the information on the community repository and in the server's source code repositories (README files and "deprecated" flag). This step assured that we focus on the current state-of-practice in LSP development.

Selection of all Java-based Servers. We selected all LSP servers implemented in Java for the following three reasons. First, substantial language engineering tooling is implemented in Java, allowing us to compare the LSP servers with our own experiences in building DSLs in Java-based tooling (specifically, EMF, the parser generator ANTLR, and other related frameworks for language analysis and program or model transformation) [6, 14, 29, 32, 42, 43, 48, 51, 57, 63, 64]. Second, Java is among the most popular and well-understood programming languages, easing our analysis (as opposed to analyzing servers written in R, Go, Erlang, or Elixir, for instance). Third, Java is, together with Typescript, the most popular implementation language among all LSP servers documented [53].

Random Selection among all other Servers. Being confident about understanding the necessary details about our Java-based LSP servers, we continued the analysis for a more diverse sample.

Table 1: Overview over the analyzed LSPs

	target lang.	type	impl. lang.	source repository
1	ActionScript3	GPL	Java	github.com/BowlerHatLLC/vscode-nextgenas/tree/master/language-server
2	R	GPL	R	github.com/REditorSupport/languageserver
3	Go	GPL	Go	github.com/golang/tools/tree/master/gopls
4	Erlang	GPL	Erlang	github.com/erlang-ls/erlang_ls
5	Flux	DSL	Rust	github.com/influxdata/flux-lsp
6	Assembler	GPL	C++	github.com/eclipse/che-che4z-lsp-for-hlasm
7	XML	DSL	Java	github.com/angelozerr/lsp4xml
8	Turtle	DSL	Type-script	github.com/stardog-union/stardog-language-servers/tree/master/packages/turtle-language-server
9	C/C++	GPL	C++	github.com/MaskRay/ccls
10	Java	GPL	Java	github.com/eclipse/eclipse.jdt.ls
11	Robot Framework	DSL	Python	github.com/robocorp/robotframework-lsp
12	Rust	GPL	Rust	github.com/rust-analyzer/rust-analyzer
13	Xtext (any lang.)	DSL	Java	github.com/eclipse/xtext-core
14	Python	GPL	Python	github.com/palantir/python-language-server
15	Cobol	GPL	Java	github.com/eclipse/che-che4z-lsp-for-cobol
16	Elixir	GPL	Elixir	github.com/elixir-lsp/elixir-ls
17	Puppet	DSL	Ruby	github.com/lingua-pupuli/puppet-editor-services
18	PHP	GPL	PHP	github.com/felixfbecker/php-language-server
19	Groovy	GPL	Java	github.com/prominic/groovy-language-server
20	LaTeX	DSL	Rust	github.com/efoerster/texlab
21	Apache Camel	DSL	Java	github.com/camel-tooling/camel-language-server
22	Ballerina	GPL	Java	github.com/ballerina-platform/ballerina-lang/tree/master/language-server
23	Java	GPL	Java	github.com/georgewfraser/vscode-javac
24	SonarLint	DSL	Java	github.com/SonarSource/sonarlint-language-server
25	Lua	GPL	Java	github.com/EmmyLua/EmmyLua-LanguageServer
26	MOCA	DSL	Java	github.com/mrglassdanny/moca-language-server
27	OCaml	GPL	OCaml	github.com/ocaml/ocaml-lsp
28	TTCN-3	DSL	Go	github.com/nokia/ntt
29	Swift & C-family	GPL	Swift	github.com/apple/sourcekit-lsp
30	Vala	GPL	Vala	gitlab.gnome.org/esodan/gvls

We randomly selected additional servers until we reached a total sample size of 30 servers, including 18 additional LSP servers.

The purpose of this sampling process was to control the selection of the LSP servers with the aim of enabling proper exploration among existing servers. Table 1 provides an overview of the studied LSP servers. We selected 30 LSPs (≈30% of the curated list), implemented in 14 and targeting 31 programming languages or DSLs. Taking saturation [25] as a quality criterion, we reached saturation after analyzing the 18th LSP server of our sample that is when we did not find any new insight (explained shortly, in Sec. 3.3).

3.2 Quantitative Analysis

We quantitatively analyzed the LSP servers to identify frequently implemented features and correlations among them. We focus on frequently implemented features as they provide us with a homogeneous range of implementations to be compared and help us to identify relevant concerns and common implementation practices.

To collect the features implemented in the servers, we took advantage of the information available by default in each LSP (i.e., following the LSP specification). Every LSP server has a method called *initialize* [35], which is the first method requested when a client starts using the server. This method returns an object implementing the interface *InitializeResult*, having the property *capabilities* that describes all features available in the server. Since many servers were hard to build and run, while nearly all servers had

the required information hard-coded, we statically analyzed their implementations. Thus, for every server, we inspected the source code of the *initialize* method to collect the features provided. We also obtained the versions of the LSP specification implemented. At the time of analysis, the version of the LSP specification was *3.16.0*, released on 12/2020. As it is a recent version and many of the LSPs in our sample did not implemented it at the time of analysis, we just considered features defined until version *3.15.0*, from 01/2020.

3.3 Qualitative Analysis

To identify frequently used implementation practices, we applied an iterative analysis approach in which we first identified relevant *concerns* [2] and related *practices* afterwards. To be more precise, as shown in Fig. 1, we proceeded as follows.

Based on the expert knowledge of the second and fourth authors, we started with two initial *concerns* focusing on the architecture and implementation-details of LSP servers. Driven by these two concerns, we systematically collected common implementation aspects for all LSP servers. First, we iteratively analyzed the individual LSP server implementations and extracted *relevant segments* (i.e., portions) of the source code that are related to the initial concerns. Thereafter, we *labeled* each relevant code statement with how exactly it addresses the initial concerns. For deriving more detailed concerns, we grouped the labeled statements based on their *commonalities*. Then, we refined these groups based on new or already identified *concerns*. We continued this process for the next LSP server until we analyzed our entire sample.

To formulate the practices addressing the identified concerns, we conducted an exploratory analysis [15], starting from the identified concerns. To this end, we analyzed the overall project structure of each LSP server and identified entry points for each considered LSP feature. After the entry point identification, we started an extensive analysis of the source code to identify implementation practices following a process inspired by thematic analysis [12]. We followed the feature implementation line-by-line but also dependencies, e.g., method calls. We captured relevant practices on a higher level as coding themes and described them as *possible values*, which served as the codes of the thematic analysis. Whenever we saw an interesting aspect of the implementation (libraries, code patterns, interaction with other features/resources), we added a new code. Then, we consolidated these codes in a group discussion to formulate practices related to the concerns.

4 CONCERNS AND PRACTICES

We identified concerns by looking into typical software implementation aspects and considering activities such as software design (i.e., architecture, including ways, how the system is structured), followed by implementation practices for realizing the planned LSP server. All concerns are specific to language engineering, e.g., we looked at how instances are represented in a server (i.e., as an AST or a simple string of characters on which the editing support is implemented). In what follows, we present, discuss, and answer each identified concern with the practices observed in our study. The first concern is based on our quantitative analysis, while the remaining six concerns emerged and were answered during our qualitative analysis.

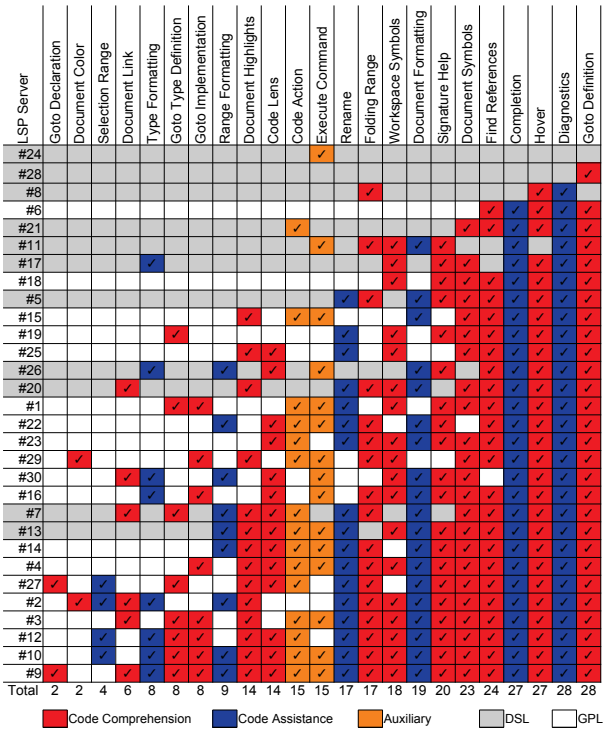

Figure 2: Features implemented in each LSP server

4.1 Concern: Selection of Editing Features

When implementing a new LSP server, it is useful to know which features are typically provided by LSP servers. To get a general overview of possible features supported by LSP servers, we quantified all features provided by the LSP servers in our sample.

4.1.1 What are the most frequently implemented features? Figure 2 shows which features are implemented by which server. LSP features are in the columns, and servers are in the rows. Both are sorted in ascending order by the number of implemented features or servers implementing the feature, respectively. We discuss the provided features according to what kind of editing support they offer and their frequency in the LSP server implementations. Understanding the frequency of the individual editing features can help engineers in deciding which features to implement first, as well as researchers in focusing their research on industry needs.

`Goto Definition` and `Diagnostic` are the most frequently implemented features. These features constitute common editing support for languages [27]. Only two servers miss these features, servers #24 and #8 for `Goto Definition` and servers #24 and #28 for `Diagnostic`. These servers are special, they use LSP as a vehicle to offer third-part services, such as offering a linter (#24).

Due to space limitations, we cannot describe all features in detail and refer to the LSP specification [35]. Next, we focus on the most frequently implemented third of LSP features, totaling seven features. Reducing the number of features enabled us to analyze their implementations in-depth. In what follows, we summarize these seven features and the servers implementing them.

(1) `Goto Definition` (*Code Comprehension*, 28 servers) allows users to quickly navigate within a project to find where a reference (e.g., a variable) is defined.
(2) `Diagnostic` (*Code Assistance*, 28 servers) returns results/errors from compilers, parsers or even lint tools.
(3) `Hover` (*Code Comprehension*, 27 servers) shows information related to a given text document position.
(4) `Completion` (*Code Assistance*, 27 servers) computes a list of completion items at a given cursor position.
(5) `Find References` (*Code Comprehension*, 24 servers) collects all references pointing to the symbol at the target position.
(6) `Document Symbols` (*Code Comprehension*, 23 servers) returns either all symbols present in a document or even the entire hierarchy of the present symbols[35].
(7) `Signature Help` (*Code Comprehension*, 20 servers) provides additional information such as what is the parameter currently selected or the method's documentation.

When investigating the seven most frequent features, we noticed that these are the more generically specified features of the list in Fig. 2, which give great flexibility on how and to what extent to implement them to the developers. Altogether, these features cover different aspects of language support ranging from resolving references (`Goto Definition` and `Find References`) over aggregating relevant information (`Hover`, `Document Symbols`, and `Signature Help`) to features providing more complicated tasks (`Diagnostic` and `Completion`). This variety indicates that editing support depends not primarily on a single kind of editing feature, but needs rather diverse features.

Three LSP servers of our sample implement noticeably few features (#8, #24, #28) and use LSP, as mentioned above, just as a basis to offer third-party services. *SonarLint* (#24) only implements `Execute Command` to forward the SonarLint [52] standalone-tool's static code analysis results. Similarly, *nokia/ntt* (#28) is an LSP server for language-agnostic testing with TTCN-3, a DSL for scripting tests. Finally, *Turtle* (#8), which is the basis of the Stardog IDE [55], only implements `Folding Range`, `Hover`, and `Diagnostics` as a wrapper for a data exploration tool. Altogether, LSP servers implementing only a few features usually use LSP to forward the functionality of powerful third-party libraries.

4.1.2 Which features to implement for a language? Previously, we investigated what are the most frequently implemented editing features and discussed possible reasons. Despite knowing what can be provided by LSP servers, due to limited resources, this is not enough information to develop an LSP server. The challenge is to decide which features to implement. Thereby, we can have two influencing factors. First, language-specific properties, such as language paradigms, can impact the feasibility of the different features. Second, there can be an interaction among the features, e.g., features complementing each other.

Relations Between Languages and LSP Features. For all servers, we calculated the correlation between implemented features and language characteristics of the target languages. We used the R implementation [49] of the Spearman's rank correlation [41]. Figure 3 shows the resulting correlation matrix with the language characteristics on the vertical axis and the features as well as the number of

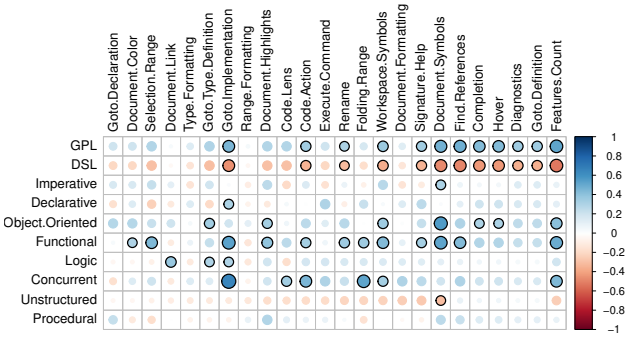

Figure 3: Correlations between editing features and paradigms of instance languages (bold circles are significant at a significance level of 5%).

implemented features on the horizontal axis. As language characteristics, we consider language type (general purpose language (GPL) and DSL) and language paradigms (e.g., imperative, declarative, or functional). Based on our sample size, all correlations with an absolute correlation higher than 0.306 are statistically significant at a level of 5% [31]. Altogether, we observed 55 significant correlations of which we discuss the most interesting ones in what follows.

First, GPLs tend to have a positive correlation with all LSP features whereas DSLs tend towards negative correlations, meaning that servers for GPLs are in general likely to implement features while servers for DSLs are likely to not implement features. We can also see this correlation in Fig. 2, where DSLs are shown rather on the top. However, popular DSLs such as XML, LaTeX, or Xtext still implement many features. Therefore, the number of implemented features could mainly depend on the language's popularity. This is supported by the popular languages Java, C/C++, and Rust that are used to implement the most LSP features.

When looking at the individual correlations, we notice that there is a strong correlation with *Functional* and *Concurrent* languages for the feature `Goto Implementation`. Practically, there is a distinguishment between signature and implementation in such languages, explaining the observed popularity. While one would also expect any *Object-Oriented* language to be supported by this feature, this is only easily possible for class-based languages but not for prototype-based languages such as Python.

Object-Oriented and *Functional* languages have a rather strong correlation with `Document Symbols`, reflecting the strong structuring in such languages and the need to assist developers by providing all language constructs contained in a file.

For all other LSP features we cannot find strong correlations in our example, either because they are implemented by nearly all LSP servers or they are implemented only in a few cases, making them more specific to individual considerations. In summary, besides features that should be implemented in nearly all LSP servers, there is some editing support specific to language paradigms. Still, the popularity of a language seems to be the main driver in implementing editing support and might also provide the resources to implement less relevant features.

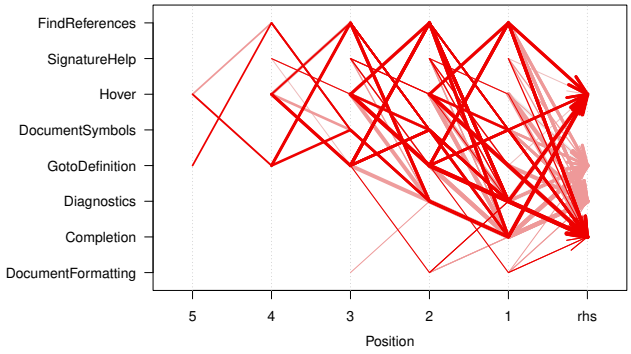

Figure 4: Parallel coordinates plot for 167 rules

Relations Among LSP Features. To identify features usually implemented together, we utilized association rule mining that discovers associations among data items [1], like if *A* and *B* are implemented, *C* is implemented most likely, too. For that, we used the algorithm Apriori, available on the R package *arules* [59]. As parameters, we set the thresholds *support* to only consider features in at least 63% of the servers, and *confidence* to 0.95, which means that the features must appear together in at least 95% of the cases.

The algorithm mined 167 associations rules, summarized in Fig. 4. Positions 1 to 5 indicate how many features are in the *antecedent*, and rhs indicates *consequent*. The thickness of the arrows represent the *support*, namely how strong is the association—thicker means stronger, and the shading the *lift*, which indicates how likely is the association to happen—darker means more likely.

By analyzing the data, we can observe: (i) there are few rules with five or four features, mostly involving `Goto Definition` and `Hover`, which are the first and the third most frequent features; (ii) there are no rules in which the *consequent* are the features `Find References`, `Signature Help`, `Document Symbols`, and `Document Formatting`; (iii) the rules with stronger *lift* have as *consequent* `Hover` and `Completion`; (iv) `Find References`, `Diagnostic`, and `Completion` are among the features with stronger *support*. Overall, these results show that when servers have the popular features `Find References`, `Diagnostic`, and `Goto Definition`, they will most likely also have `Hover` and `Completion`.

4.2 Concern: Use of Third-Party Libraries

Libraries are a common practice for sharing and reusing functionalities [26]. In some cases, even the entire language support can be realized by wrapping libraries. Therefore, we are interested in the roles of external libraries in the LSP implementations and searched for recurring kinds of libraries.

4.2.1 Commonly Used Libraries. For many recurring problems, libraries provide mature solutions. We identified six different kinds of libraries that are frequently used in the LSP server implementations.

Compiler/Parser (25 servers). Most frequently, LSP servers use compilers or parsers to create a document's abstract representation, which they traverse to collect required information, e.g., reference positions or suggested refactorings, such as renaming a symbol. One example is the library Tolerant PHP Parser [34].

Parser Generators and Maintaining a Grammar (3 servers). Comparable to compilers or parsers, custom grammars allow the creation of an abstract syntax representation from a document's plain contents, but the grammar that describes the tokens of the target language is part of the LSP project and used to generate a parser (e.g., using Antlr [40]). Besides the LSP features, the LSP server has to maintain the grammar, e.g., when language constructs are introduced or deprecated as part of the target language's evolution. For instance, the server *MOCA* (#26) maintains a grammar for the DSL MOCA (see Table 1).

Documentation Handler (5 servers). Documentation such as JavaDoc is usually declared in another syntax as the target instance. It is ignored by parsers and can even be stored separate documents. Libraries such as Jedi [28] for Python, ElixirSense [20] for Elixir, and DocBlock [45] for PHP can handle such documentation and map it to corresponding instance elements.

Environment Info Handler (9 servers). Besides information from the documents and the language specification, environment-specific information (e.g., libraries or the installed language environment) can be beneficial for features such as `Code Completion`. For instance, the libraries M2Eclipse [19] and Buildship [16] can collect information on Maven and Gradle dependencies of Java projects.

Linter (3 servers). The LSP feature *Diagnostics* provides information such as wrong syntax. In addition to compiler reports, linters and analysis tools are used to collect diagnostics. For instance, the library pytlint [47] is used to validate the files and return warnings related to source code formatting, the presence of smells, or recommended refactorings (server #14). The server *Elixir* (#16) uses the static analysis tool Dialyzer [22] to provide similar diagnostics.

LSP SDK (13 servers). There are a few SDKs helping engineers to deal with LSP-related details (e.g., schemata of messages). We found usages of LSP4J [18] for server implementations written in Java as well as *Microsoft/vscode-languageserver-node* [36] targeting servers implemented in Javascript/Typescript and running in Node.js.

In summary, libraries are mainly used to avoid custom implementations for accessing the document instances or additional needed information for realizing LSP features.

4.2.2 Wrapped Language Support Libraries. While libraries mainly support LSP feature implementations, 7 servers benefit from libraries that implement LSP features entirely or partially. For example, Eclipse JDT [17] is a library used by the *Java* server (#10) that implements features including `Completion` and `Find References`. We observed the same with the library Jedi [28] used by the *Python* server (#14). Those libraries provide enough features so that the server becomes a wrapper around them, which only needs to pass the document to the library and wrap its response, respectively.

Considering the decreased effort when utilizing libraries to implement any aspect of an LSP server, engineers still have to consider extra efforts to translate interfaces and wrap responses to utilize library (e.g., translating linter diagnostic messages). Defining interfaces to encapsulate and avoid the libraries' interfaces polluting the server interfaces, potentially using design patterns (e.g., *Adapter* or *Facade* [24]), must also be considered by LSP developers.

4.3 Concern: Structuring LSP Servers

Architecture plays an essential role in software engineering and is crucial for the successful development and maintenance of software systems [4, 24]. In the literature, various architectural patterns and

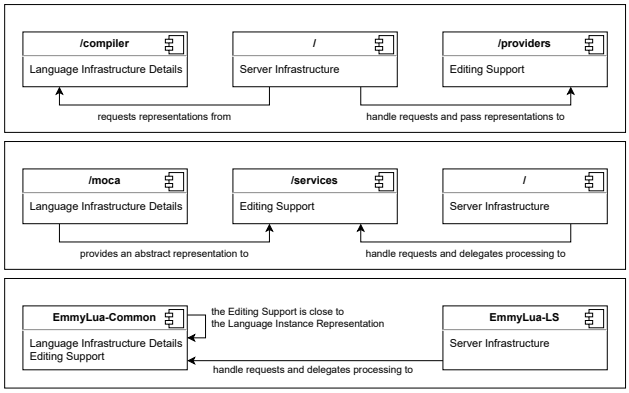

Figure 5: Example of server layers observed

principles [24, 58] have been proposed, but when incorrectly applied, anti-patterns [8, 44] can emerge that can challenge realizing a software system. Accordingly, we have to identify suitable design principles for LSP servers.

When investigating the source code of the LSP servers, we identified two design principles that were applied or are applicable to all investigated LSP server implementations. First, the architecture of LSP servers can be realized based on a layered architectural style. Second, LSP servers must consider the overall flow to completely implement a feature when deciding how to structure the internal communication of the components.

4.3.1 Layered Architecture: We observed that LSP servers are mainly structured into three layers (*Server Infrastructure*, *Language Infrastructure*, and *Editing Support*) with different responsibilities for the LSP feature execution.

Figure 5 shows examples of the layered structure of LSP servers. The server *Moca* (#26) is structured into well-defined package. The package *moca* contains code to deal with the *Language Infrastructure* details, such as parsing and creating an abstract instance representation, the package *services* the actual implementation of *Editing Support* capabilities, and the project root contains code to deal with the *Server Infrastructure*. A similar packaging structure is followed by *Groovy* (#19) and *Latex* (#20).

However, not every server follows this structure. For example, *Lua* (#25) has two main components: (i) *EmmyLua-Common* that aggregates the *Language Infrastructure* and *Editing Support*, and (ii) *EmmyLua-LS* providing *Server Infrastructure*.

Altogether, the responsibilities of the layers are as follows:

Language Infrastructure. This is the lowest layer and implements the immediate interaction with the documents (e.g., parsing the source code for creating an abstract syntax representation). If needed, support for multiple languages can be provided within this layer. The input for this layer is usually a document and the output is an instance representation that can easily be processed to realize LSP features, such as an abstract syntax tree (AST). As the parsing process can generate additional relevant information (e.g., warnings/errors observed at parsing or compilation), it can also implement patterns such as *Publish/Subscribe* to provide such information to the *Editing Support* layer above it.

Editing Support. This layer realizes the feature-specific processing. For example, traversing the abstract representation to collect completion items. Some servers opted to delegate this layer's responsibility to language support library that provide features including *Rename* and *Completion* (cf., Sec. 4.2).

Server Infrastructure. This layer is responsible for dealing with protocol details (e.g., handling feature requests). Some protocol details can be delegated to an LSP SDK. A curated list of LSP SDKs for different languages can be found at the official LSP website,[1] and we discuss observed examples in Sec. 4.2.

Even when the servers encapsulate code in the same layers, the information flow might differ. Therefore, in what follows, we discuss the most commonly observed execution flow.

4.3.2 Feature Execution Flow: For the servers not only wrapping libraries, we identified the feature execution flow described in Sec. 2. After receiving a request, the server loads the document into memory and then calls for each feature a provider that handles it. The feature execution is usually implemented in three steps:

Parse. Navigating through a document's symbols is usually needed to implement features such as *Completion* and *Find References*. To this end, most of the servers analyzed need to parse the document into an abstract instance representation in the first execution step.

Traverse. The abstract syntax representation is then traversed to collect the elements that will be needed to process the feature (e.g., all AST nodes that are symbols in a specific document).

Process. The collected elements are then processed to realize the respective LSP feature. This can involve filtering, ordering, or collecting additional information that is required to process the feature.

Besides this being the most common sequence, specific cases where the execution of a feature is delegated to a library may result in a different sequence of steps. One example is the feature *Diagnostic*, which usually returns results or errors collected from a compiler, parser, or even a linter library. The LSP server just translates the messages into the response format required by the client, making all other execution steps within the LSP server obsolete.

The observed flow is predestinated to applying the Pipes and Filter pattern. However, while we found indications of this pattern, no LSP server strictly follows the pattern. Altogether, LSP server implementations can be effectively structured into layers containing task-specific filters that can be flexibly exchanged (e.g., to support multiple instance languages).

4.4 Concern: Implementation Granularity

When implementing an LSP server, one has to decide how granular the implementation of features should be, and whether it makes sense to separate feature implementations (and for which features) into separate modules. We investigated every LSP server implementation and assigned a granularity level based on a 5-level Likert scale ranging from the most coarse-grained (1) to more fine-grained (5) granularity of the implementation:

1: Single Monolith (1 server). This granularity describes servers that implement all provided features in a single structuring entity (e.g., only one class). Just *SonarLint* (#24), which does not implement any feature on its own, is implemented at this granularity level.

[1] https://microsoft.github.io/language-server-protocol/implementors/sdks/

2: Delegator per LSP Feature (6 servers). At this granularity a single file that consists out of delegator methods is used to call a library that will effectively implement the feature. This was the case for almost all servers that reuse *Language Supporting Libraries*, such as the *Python* server (#14). The only server that uses *Supporting Libraries*, but that is not on this granularity is *Swift* (#29), which besides delegating the execution to external libraries, also implements support for multiple languages (level 5).

3: File/Class per LSP Feature (13 servers). The most common case is where the server has one file or class per feature with encapsulated methods/functions to implement details of the features (e.g., traversal strategies or even validation of the context where a symbol applies).

4: Module of Multiple Files/Classes per LSP Feature (8 servers). The features' implementation details are well encapsulated in multiple files/classes as the implementation's complexity requires additional structuring. Among others, those feature implementations contain non-trivial searches for specific content or provide contexts to select AST nodes, e.g., variable/function completion in *Ballerina* (#22), or even have their own set of possible values, e.g., snippets/keywords completions in *Robot* (#11).

5: Multiple Modules per Target Language (2 servers). At this granularity, each feature is implemented in multiple modules (e.g., because the server provides mechanisms to implement the features for multiple target languages). For example, *Turtle* (#8) is one of the supported languages in the server *Stardog*. The support for all its target languages is implemented as extensions of an *AbstractLanguageServer* that orchestrates the language-specific feature implementations execution. The editing support for each target language is encapsulated in a module that must provide methods to parse documents and provide diagnostics when changes happen. While the server *SourceKit* (#29) is based on a *Language Supporting Library*, it also implements the logic to delegate the feature execution to multiple language-specific libraries (e.g., the *Swift* programming language) and translates the result to the expected response format (e.g., a *HoverResponse*).

The servers tend to have straightforward implementations with one file per feature as the most common granularity. We can observe a tendency to finer-grained structuring when the features of LSP servers get more complicated, e.g., when a feature is implemented to support multiple languages.

4.5 Concern: Language Instance Representation

For providing editing support, it usually necessary to have an intermediary representation of the documents. While we observed that in some cases LSP servers can immediately operate on concrete syntax (e.g., source code representations), for most tasks a structured instance representation in abstract syntax is needed.

4.5.1 Abstract Syntax Representation. To make information about the program or model instances accessible, most LSP servers (25 servers) use ASTs. An AST is a structured, tree-based representation of the concrete syntax, in which nodes of the tree represent the symbols and the edges are the relations between the symbols. Also, simpler representations, such as the Domain Object Model (DOM) are used (e.g., in the R server (#2)), or servers immediately work on the plain documents (e.g., SonarLint (#24)).

4.5.2 Tree Traversal. From identifying a symbol or even the parameters of a function node to collecting all variables declared in a document, traversing the abstract representation of a document is a very common task realized in LSP feature implementations. This traversal can follow different strategies, from standard graph traversal algorithms to ad hoc implementations for specific cases to design patterns and third-party code.

As abstract syntax representations are usually instances of trees or graphs, the most common strategy (13 servers) is to traverse them using standard algorithms (e.g., breadth first search). Among others, this strategy is used to pre-process the document structure (*Go* server, #3) and create auxiliar index structures (cf. Sec. 4.6) or to collect nodes when implementing a feature such as `Find References`.

Simple ad hoc algorithms for navigation are used by 10 servers. for instance, the feature `Completion` in the *XML* server (#7) simply navigates all parent nodes to suggest completions. Another usage of such a bottom-up navigation is identifying the scope in which a symbol is declared (e.g., whether a variable is below a function node or a class definition). Finally, ad hoc traversal implementations are also used to determine if the cursor position is within some specific semantic structure, such as the right-hand side of an assignment statement (*Flux* server, #5).

To make the traversal implementation reusable for multiple features without implementing it each time, the *Visitor* pattern [24] was used in 10 servers. The possibility to implement server-specific visitors is commonly provided by compilers or language support libraries (e.g., *javac*'s *TreePathScanner* [38], used by the server #23). LSP servers use visitors to filter tree nodes or to accept nodes defined by the visitor logic when running the feature `Completion` (*C/C++* server, #9). Besides object-oriented implementations, other variations utilize lambda functions (e.g., *rust-analyzer* uses Hir [60] to collect expected results). Some libraries like *JDT* and *Clang/sema* go one step further and provide declarative ways to specify node characteristics for which the libraries directly execute features such as `Completion`, `Hover`, or `Find References`. Lastly, we found cases of using XPath for navigation, such as in the *R* server (#2).

Altogether, many libraries provide sophisticated support for representing and traversing documents for all major languages. If no suitable library is available for a target language, the needed functionality can be generated after providing a grammar of the target language (e.g., using *Xtext*). Only when using libraries is not possible, or when there are good reasons for not using a library, LSP server developers need to implement the abstract representation as well as the traversal on their own.

4.6 Concern: Performance

For providing efficient and usable editing support, response times are critical for the document editing experience by users. Consequently, we expected to observe optimization strategies (e.g., caching) to improve the features' execution performance. For example, avoiding constantly re-parsing documents can increase performance. We identified three common optimization strategies:

AST Caching. To avoid frequent recreation of abstract representations, these are stored in memory with a key-value relation that

matches the file URI and the data structure (e.g., the root node of an AST). The data structure can be loaded when the user notifies that the file was opened (like in the *XML* server, #7) or even in the server's startup phase where all files of the workspace are compiled and cached. Here, it is required to implement mechanisms for updating the abstract representation after changes to the document.

Auxiliary Index Structure. A strategy used by 13 servers to access a representation of the document but avoiding direct operations on the abstract representation is to provide an auxiliar index structure that contains relevant nodes of the parsed document. The strategy is to once collect relevant nodes from the abstract representation and store them in a navigable data structure. One concrete representation of such an auxiliary index structure is a symbol table. As the indexed structure stores the information about the symbols within a document, it can simplify the implementation of LSP features. Developers do not need to explicitly implement the traversal of the representation and to differentiate between the different node types for each feature implementation. In addition, it can also increase performance as information collected once can be reused for future feature calls. As with the abstract representations, this indexed structure needs to be reprocessed every time a document is changed (e.g., as implemented in the *Cobol* server, #15).

Enclosing Scope Scanning. In addition to using an abstract representation to implement LSP features, 22 servers also directly operate on a document for specific cases, avoiding the overhead of constructing additional data structures. In principle, simple string operations relying on navigating the text character by character can be used to scan the document and collect information (e.g., where the current symbol begins). This information can also be gathered by regular expressions, applied to either, the entire document or just the target line, like the *Elixir* server (#16) does. As such information can usually be collected from the AST node attributes there is no need for explicitly processing the plain document.

Some servers check syntactic correctness through the direct analysis of the documents by relating a specific position in a document to other positions (e.g., to identify whether a symbol is located within any brackets or not). Using such information, the feature implementation can determine if the symbol is inside the parameters section of a method declaration or a function call (*Xtext* server, #13). Other applications of this kind of logic is to identify if a special character is preceding the current symbol (*Assembler* server, #6) or to identify if the cursor is within a comment block (*R* server, #2).

Altogether, optimizations mainly focus on reducing the run-time overhead for building intermediate document representations as well as for searching these. As a downside, developers need to handle updating the cached data when document changes emerge, which is the subject of the next concern.

4.7 Concern: Handling Instance Changes

When providing editing support for languages, the constant change of documents is omnipresent and has to be considered to preserve the validity of the feature executions. The LSP protocol already defines that the server must be notified when changes happen or documents are saved [35]. Any performance optimization, such as in-memory abstract representations and even index structures must be reprocessed when they do not reflect the state of the related resource anymore. The LSP specification establishes the *textDocument/didChange* and *textDocument/didSave* requests to notify the LSP server that the resources were updated.

Among our servers, the most common case (19 servers) of synchronization is the incremental application of the changes to the resources, while nine servers synchronize the text based on the full content. The servers *SonarLint* (#24) and *Turtle* (#8) do not update the text content, but reload the entire content at every feature call.

The strategies to update the representations with the changed content are (i) *reprocess* the file, parsing it again and recreating the abstract representation; (ii) *invalidate* the file, marking it as stale or cleaning the in-memory representation, so the file will be reprocessed when the next feature is requested. In our sample, 13 servers reprocess upon *textDocument/didChange* requests, while seven invalidate. Upon *textDocument/didSave* requests, five servers reprocess and two servers invalidate the document.

In summary, handling document changes is essential for properly providing editing support. We identified two strategies for tracking document states, identifying the need to update caches, and actually reacting to document changes.

5 DISCUSSION & LESSONS LEARNED

Recall that providing good editing support requires implementing very different kinds of editing features. As such, our results on what are the most implemented editing features, together with their relations to language paradigms, can guide LSP developers when prioritizing the editing features.

Interestingly, one does not always need heavyweight and full-fledged language processing implementations. For instance, we observed that using abstractions other than ASTs for representing language instances can be sufficient. Also, auxiliary index structures can avoid manual traversals of the AST for each feature request. We observed that tree traversal is implemented repeatedly for different purposes and has a high potential for reuse. However, this might lead to more complicated server architectures than the ones that we observed in our study.

The layered architecture proposed by us can easily support such reusable mechanisms and allows flexibly connecting different implementations. Providing more sophisticated editing support, e.g., supporting multiple languages or more sophisticated analyses will result in a shift from one class per feature, as mainly observed in our study, toward multiple classes per feature. Further modularizing LSPs is still an open research problem.

In general, libraries help to implement editing support, but cannot do all the work and even generate additional concerns such as how to encapsulate the external library's interfaces to avoid the pollution of the project interfaces. In this regard, generated editing support can be one way to go for standard editing support. Most servers for DSLs (as opposed to the servers for programming languages) use ANTLR (4 servers), but only for parser generation. The editing support implementation is hand-written, similar to the other LSPs. Only the LSP realization of Xtext generically provides editing support for any Xtext-based language, making the implementation more fine-grained than the hand-written ones. It does not target a specific language, but any that can be expressed in an Xtext grammar. For instance, in a *Completion* request, possible symbols are looked up from

the grammar. In the end, the generated editing support implementation is the same as when Xtext generates an editor plugin for Eclipse.

All the languages supported by our language servers are so-called parser-based languages. A complementary paradigm is projectional editing (a.k.a., syntax-directed or structured editing) [6, 62], where a user's editing gestures directly change the underlying AST, without any parsing involved. The core benefits are basically unlimited language composition and flexible notations (textual and visual ones). While we did not observe any specific support, the fact that LSPs have been developed for graphical modeling languages, such as Eclipse GLSP,[2] shows that LSP can be used for projectional-editing-based languages, such as implemented in Jetbrains Meta Programming System (MPS). The main concern is to represent and pass the location of elements edited, which cannot be based on a line number and character number pair, but requires some other locator mechansms (e.g., node ID).

6 THREATS TO VALIDITY

External Validity. First, our focus on LSP to investigate editing support might bias our results unnecessarily towards LSP technicalities, limiting generalization to editing support in general. Still, LSP is a widely used protocol for providing tool-independent editing support, and the communication between clients and the LSP server is only a minor part of the server implementations. Second, our selection of LSP servers could have biased our findings (concerns and solutions). To mitigate this threat, we followed a four-step selection process to retrieve a suitable sample. We also cover a good range of languages, the sample of servers we studied provides support for 34 different languages. Our server sample also relies on 14 different implementation languages, reducing the bias of our findings. Third, our results could be biased towards either DSLs or programming languages. However, our sample contains 11 of the former and 19 of the latter, reducing bias.

Internal Validity. First, during our analysis, we might have misunderstood some LSPs' internals. However, our selection strategy assured that we first investigate Java-based servers, relying on our experience with Java-based language engineering tooling. Second, we started to look into architecture and then implementation, specifically, our starting points were the seven most frequently implemented features. This selection could miss important practices only seen in the other features. However, our goal of supporting other developers when implementing their own LSP server or "popular" features justifies this scope and potential bias. Third, we used the well-established thematic analysis method. After collecting the data of each server, the authors extensively discussed the codes, labels, and refined the concerns.

Conclusion Validity. Even tough, the implementation practices identified and discussed in this work give detailed insights into how to implement editing support, these could be considered as somewhat limited to the observed frequencies of the practices along the concerns analyzed in Sec. 4 and Sec. 5. Still, the work builds a solid foundation for further assessment of which practices work best for the least amount of effort, and which practices go hand in hand across the considered concerns.

[2] https://www.eclipse.org/glsp/

7 RELATED WORK

Rodriguez-Echeverria et al. [50] propose an LSP infrastructure for graphical modeling languages. Their goal is decoupling the editing support from the graphical language. To this end, they studied different alternatives to the LSP. In contrast, our work focus on more general practices for implementing LSP servers.

Erdweg et al. [21] present details of language workbenches for developing GPLs and DSLs. They investigate the implementation of 10 language workbenches regarding feature coverage, size, and required dependencies. However, nothing is explored concerning editing support. Additionally, their work is somewhat dated (from 2013) regarding recent advances in language engineering and editing support. Recall that LSPs appeared in 2017.

Bünder and Kuchen [10] present how the LSP can be used to satisfy both developers and domain experts, with different preferences, while working on the same projects. LSP provides the means for integrating different editors in model-driven software development projects. In a recent book [27], the developers of the Ballerina Language Server (a cloud-native programming language) describe the requirements to implement editing support using LSP. Then, they detail the implementation for the client and the server of the Ballerina language. Both works are experience reports for a specific scenario or technology stack, which is different from our goal of systematically studying implementation practices in many servers.

Stolpe et al. [56] describe the use of LSP to implement code editing that is reusable for any IDE with LSP support. However, they focused exclusively on the Truffle framework as a target language. In contrast, we describe details of implementation practices, considering several types of target languages. Mészáros et al. [37] also leverage LSP to integrate a code comprehension tool named CodeCompass as part of Visual Studio Code. With a similar purpose, Pupo et al. [46] use LSP to integrate a machine-learning-based tool for improving Javascript security into an IDE. These papers mainly describe the implementations for these specific cases without discussing concerns or practices that must be taken into account.

8 CONCLUSION

Effective editing support for software languages is crucial. We presented the first set of engineering practices, systematically identified—quantitatively and qualitatively— from a sample of 30 LSP servers. We synthesized seven core concerns and present practices for realizing language editing support. We found that a variety of features are required to provide editing support, and the concrete aspects of languages play a rather minor role in the basic editing support. Still, for advanced editing support, characteristics of the target language become more important. For designing LSP servers, we identified design and implementation practices that we hope to extend to a reference architecture in later works. When it comes to the concrete implementation-level realization of LSP servers and editing support in general, for many implementation aspects, mature libraries exist and are already widely used. In addition, we observed multiple frequently used optimization techniques that usually come together with the challenge of updating a caching structure. Also, to address this challenge, we identified ways to represent the language instances, to traverse these representations, and to handle changes to them, which must trigger updates of these representations.

REFERENCES

[1] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast Algorithms for Mining Association Rules. In *20th International Conference on Very Large Data Bases (VLDB)*, Vol. 1215. 487–499.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.*

[3] Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. 2022. *Supplementary Material – Editing Support for Software Languages: Implementation Practices in Language Server Protocols 59.* https://doi.org/10.5281/zenodo.6974153

[4] Len Bass, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice.* Addison-Wesley Professional.

[5] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen Vinju. 2015. Modular Language Implementation in Rascal – Experience Report. *Science of Computer Programming* 114 (2015), 7–19. https://doi.org/10.1016/j.scico.2015.11.003 LDTA (Language Descriptions, Tools, and Applications) Tool Challenge.

[6] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE).*

[7] M. Bialy, V. Pantelic, J. Jaskolka, A. Schaap, L. Patcas, M. Lawford, and A. Wassyng. 2017. Software Engineering for Model-based Development by Domain Experts. In *Handbook of System Safety and Security*, Edward Griffor (Ed.). Syngress, 39–64.

[8] William H. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley & Sons.

[9] Hendrik Bünder. 2019. Decoupling Language and Editor-the Impact of the Language Server Protocol on Textual Domain-specific Languages.. In *MODELSWARD.* 129–140.

[10] Hendrik Bünder and Herbert Kuchen. 2020. Towards Multi-editor Support for Domain-specific Languages Utilizing the Language Server Protocol. In *Model-Driven Engineering and Software Development*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić (Eds.). Springer International Publishing, Cham, 225–245.

[11] S Cook, C Bock, P Rivett, T Rutt, E Seidewitz, B Selic, and D Tolbert. 2017. Unified Modeling Language (uml) Version 2.5. 1. *Object Management Group (OMG), Standard* 12 (2017).

[12] Daniela S. Cruzes and Tore Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *International Symposium on Empirical Software Engineering and Measurement.* 275–284.

[13] Swaib Dragule, Thorsten Berger, Claudio Menghi, and Patrizio Pelliccione. 2021. A Survey on the Design Space of End-user-oriented Languages for Specifying Robotic Missions. *Software and Systems Modeling* (Feb. 2021). https://doi.org/10.1007/s10270-020-00854-x

[14] Swaib Dragule, Thorsten Berger, Claudio Menghi, and Patrizio Pelliccione. 2021. A Survey on the Design Space of End-User Oriented Languages for Specifying Robotic Missions. *International Journal of Software and Systems Modeling* 20, 4 (2021), 1123–1158.

[15] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering.* Springer, 285–311.

[16] Eclipse Foundation. 2022. *Eclipse Buildship: Gradle integration for the Eclipse IDE.* https://github.com/eclipse/buildship Accessed: 2022-04-10.

[17] Eclipse Foundation. 2022. *Eclipse Java development tools (JDT).* https://www.eclipse.org/jdt/ Accessed: 2022-04-10.

[18] Eclipse Foundation. 2022. *Eclipse LSP4J.* https://github.com/eclipse/lsp4j Accessed: 2022-04-10.

[19] Eclipse Foundation. 2022. *M2Eclipse.* https://www.eclipse.org/m2e/ Accessed: 2022-04-10.

[20] Elixir Language Server Protocol. 2022. *ElixirSense.* https://github.com/elixir-lsp/elixir_sense Accessed: 2022-04-10.

[21] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *International Conference on Software Language Engineering (SLE).*

[22] Ericsson AB. 2022. *Dialyzer User's Guide.* https://www.erlang.org/doc/man/dialyzer.html Accessed: 2022-04-10.

[23] Martin Fowler. 2010. *Domain-specific Languages.* Pearson Education.

[24] Erich Gamma, Richard Helm, Ralph Johnson, Ralph E Johnson, John Vlissides, et al. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Pearson Deutschland GmbH.

[25] B.G. Glaser. 1978. *Theoretical Sensitivity: Advances in the Methodology of Grounded Theory.* Sociology Press. https://books.google.at/books?id=73-2AAAAIAAJ

[26] Martin L Griss. 1993. Software reuse: From library to factory. *IBM systems journal* 32, 4 (1993), 548–566.

[27] Nadeeshaan Gunasinghe and Nipuna Marcus. 2022. *Language Server Protocol and Implementation.* Apress. https://doi.org/10.1007/978-1-4842-7792-8

[28] Dave Halter. 2022. *Jedi - an awesome autocompletion, static analysis and refactoring library for Python.* https://jedi.readthedocs.io/en/latest/ Accessed: 2022-04-10.

[29] Regina Hebig, Christoph Seidl, Thorsten Berger, John Kook Pedersen, and Andrzej Wasowski. 2018. Model Transformation Languages Under a Magnifying Glass – A Controlled Experiment with Xtend, ATL, and QVT. In *26th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE).*

[30] Ralf Lämmel. 2018. *Software Languages.* Springer.

[31] Peter M Lee. 2005. Upper Critical Values for Spearman's Rank Correlation Coefficient R_s. https://www.york.ac.uk/depts/maths/tables/

[32] Max Lillack, Thorsten Berger, and Regina Hebig. 2016. Experiences from Reengineering and Modularizing a Legacy Software Generator with a Projectional Language Workbench. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC).*

[33] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and How to Develop Domain-specific Languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[34] Microsoft. 2022. *An early-stage PHP parser designed for IDE usage scenarios.* https://github.com/microsoft/tolerant-php-parser Accessed: 2022-04-10.

[35] Microsoft. 2022. *Language Server Protocol.* https://microsoft.github.io/language-server-protocol/ Accessed: 2022-04-05.

[36] Microsoft. 2022. *VSCode Language Server - Node.* https://github.com/microsoft/vscode-languageserver-node Accessed: 2022-04-10.

[37] M. Mészáros, M. Cserép, and A. Fekete. 2019. Delivering Comprehension Features into Source Code Editors through LSP. In *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* 1581–1586.

[38] Oracle. 2022. *Class TreePathScanner.* https://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/util/TreePathScanner.html Accessed: 2022-04-10.

[39] Terence Parr. 2009. *Language Implementation Patterns: Create Your Own Domain-specific and General Programming Languages.* Pragmatic Bookshelf.

[40] Terence Parr. 2022. *ANTLR (ANother Tool for Language Recognition).* https://www.antlr.org/ Accessed: 2022-04-10.

[41] Karl Pearson. 1896. VII. Mathematical contributions to the theory of evolution. III. Regression, heredity, and panmixia. *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character* 187 (1896), 253–318.

[42] Sven Peldszus. 2022. *Security Compliance in Model-driven Development of Software Systems in Presence of Long-Term Evolution and Variants.* Springer. https://doi.org/10.1007/978-3-658-37665-9

[43] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. 2015. Incremental Co-Evolution of Java Programs based on Bidirectional Graph Transformation. In *Principles and Practices of Programming on The Java Platform (PPPJ)*, Ryan Stansifer and Andreas Krall (Eds.). ACM, 138–151. https://doi.org/10.1145/2807426.2807438

[44] Sven Peldszus, Géza Kulcsár, Malte Lochau, and Sandro Schulze. 2016. Continuous Detection of Design Flaws in Evolving Object-oriented Programs Using Incremental Multi-pattern Matching. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE).* 578–589. https://doi.org/10.1145/2970276.2970338

[45] phpDocumentor. 2022. *ReflectionDocBlock.* https://github.com/phpDocumentor/ReflectionDocBlock Accessed: 2022-04-10.

[46] Angel Luis Scull Pupo, Jens Nicolay, Kyriakos Efthymiadis, Ann Nowé, Coen De Roover, and Elisa Gonzalez Boix. 2019. Guardiaml: Machine Learning-assisted Dynamic Information Flow Control. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 624–628.

[47] Python Code Quality Authority. 2022. *Pylint.* https://pypi.org/project/pylint/ Accessed: 2022-04-10.

[48] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. 2019. Geoscenario: An Open DSL for Autonomous Driving Scenario Representation. In *30th IEEE Intelligent Vehicles Symposium (IV).*

[49] RDocumentation. 2022. *cor: Correlation, Variance and Covariance (Matrices).* https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/cor Accessed: 2022-04-10.

[50] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. 2018. Towards a Language Server Protocol Infrastructure for Graphical Modeling. In *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Copenhagen, Denmark) *(MODELS '18).* Association for Computing Machinery, New York, NY, USA, 370–380. https://doi.org/10.1145/3239372.3239383

[51] Simon Schauss, Ralf Lämmel, Johannes Härtel, Marcel Heinz, Kevin Klein, Lukas Härtel, and Thorsten Berger. 2017. A Chrestomathy of DSL Implementations. In *10th ACM SIGPLAN International Conference on Software Language Engineering (SLE).*

[52] SonarSource. 2022. *SonarLint Website.* https://www.sonarlint.org/ Accessed: 2022-04-10.
[53] Sourcegraph. 2022. *Langserver.org: A community-driven source of knowledge for Language Server Protocol implementations.* https://langserver.org/ Accessed: 2022-04-05.
[54] Diomidis Spinellis. 2001. Notable Design Patterns for Domain-specific Languages. *Journal of Systems and Software* 56, 1 (2001), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3
[55] Stardog Union. 2022. *StarDog IDE.* https://www.stardog.com/studio/ Accessed: 2022-04-10.
[56] Daniel Stolpe, Tim Felgentreff, Christian Humer, Fabio Niephaus, and Robert Hirschfeld. 2019. Language-independent Development Environment Support for Dynamic Runtimes. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 80–90. https://doi.org/10.1145/3359619.3359746
[57] Daniel Strüber, Sven Peldszus, and Jan Jürjens. 2018. Taming Multi-Variability of Software Product Line Transformations. In *Fundamental Approaches to Software Engineering (FASE) (Lecture Notes in Computer Science, Vol. 10802)*, Alessandra Russo and Andy Schürr (Eds.). Springer, 337–355. https://doi.org/10.1007/978-3-319-89363-1_19
[58] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. 2010. *"Software Architecture - Foundations, Theory, and Practice"*. Wiley.
[59] The Comprehensive R Archive Network. 2022. *arules: Mining Association Rules and Frequent Itemsets.* https://cran.r-project.org/web/packages/arules/index.html Accessed: 2022-04-10.
[60] The Rust Foundation. 2022. *Guide to Rustc Development: The HIR.* https://rustc-dev-guide.rust-lang.org/hir.html Accessed: 2022-04-10.
[61] M Völter, S Benz, C Dietrich, B Engelmann, M Helander, LCL Kats, E Visser, and GH Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-specific Languages.* M Volter / DSLBook.org.
[62] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-friendly Projectional Editors. In *7th ACM SIGPLAN International Conference on Software Language Engineering (SLE).*
[63] Markus Völter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient Development of Consistent Projectional Editors Using Grammar Cells. In *International Conference on Software Language Engineering (SLE).*
[64] Andrzej Wasowski and Thorsten Berger. 2022. *Domain-specific Languages: Effective Modeling, Automation, and Reuse.* http://dsl.design