Explorando a *Program Structure Interface* (PSI): Fundamentos da Construção de *Plugins* no IntelliJ

Reinaldo Wendt[1], **Ana Carolina Rodrigues**[1], **Elder Rodrigues**[1]

[1]Laboratory of Empirical Studies in Software Engineering (LESSE)
Universidade Federal do Pampa (UNIPAMPA) – Alegrete – RS – Brasil

{reinaldowendt,anapoltronieri}.aluno@unipampa.edu.br,
elderrodrigues@unipampa.edu.br

Abstract. *This paper explores the Program Structure Interface (PSI) as a basis for plugin development in IntelliJ. Adopting a Design Science Research approach, two artifacts were implemented: the Editor Context Info, which correlates the cursor position with the hierarchy of the code, and the PSI Tree Generator, which generates visualizations of the PSI tree for source files. The results demonstrate the applicability of PSI in real-time code analysis and manipulation, highlighting its potential as a resource to support programmers and as a platform for innovation in IDEs, in comparison to the AST.*

Resumo. *Este artigo explora a Program Structure Interface (PSI) como base para o desenvolvimento de plugins no IntelliJ. Adotando a abordagem de Design Science Research, foram implementados dois artefatos: o Editor Context Info, que correlaciona a posição do cursor com a hierarquia do código, e o PSI Tree Generator, que gera visualizações da árvore PSI de arquivos. Os resultados demonstram a aplicabilidade da PSI na análise e manipulação de código em tempo real, destacando seu potencial como recurso de apoio ao programador e como plataforma para inovação em IDEs, em comparação à AST.*

1. Introdução

O desenvolvimento de software moderno demanda ferramentas que vão além da simples edição de texto [Leite et al. 2019]. Ambientes de Desenvolvimento Integrado (IDEs) tornaram-se peças centrais no ciclo de vida das aplicações, oferecendo recursos como refatoração automática, verificações estáticas e sugestões inteligentes [Golubev et al. 2021]. Para viabilizar tais funcionalidades, é necessário que o código-fonte seja representado de maneira estruturada, de modo que a IDE consiga compreender sua organização sintática e semântica [Fowler 2019]. Nesse contexto, destaca-se o *Program Structure Interface*, um mecanismo presente na plataforma IntelliJ que possibilita a interação direta com representações estruturais do código [JetBrains 2025f].

O IntelliJ, em particular, disponibiliza uma arquitetura extensível baseada em *plugins*, permitindo que desenvolvedores explorem abstrações como a PSI para criar novas funcionalidades. Essa possibilidade motiva o presente estudo, uma vez que compreender e utilizar a PSI não apenas amplia a eficiência no desenvolvimento de soluções para a própria IDE, como também serve de exemplo prático sobre como abstrações de código podem ser empregadas em tarefas de apoio ao programador [JetBrains 2025b].

Diante do exposto, o objetivo central deste artigo é investigar o potencial da PSI no desenvolvimento de *plugins* para o IntelliJ. A metodologia será baseada na implementação de artefatos que exemplificam como essa interface pode ser explorada para criar funcionalidades de análise e manipulação de código. Dessa forma, o trabalho busca oferecer evidências da aplicabilidade da PSI em cenários de apoio inteligente à programação.

Além desta introdução, o artigo está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica, discutindo representações estruturais de código, a importância de abstrações em IDEs e a definição da PSI. A Seção 3 descreve a metodologia adotada. Na sequência, a Seção 4 detalha os *plugins* construídos e seus aspectos técnicos. A Seção 5 apresenta a análise dos resultados e das contribuições. Por fim, a Seção 6 traz as considerações finais, destacando limitações e perspectivas de trabalhos futuros.

2. Fundamentação Teórica

Esta seção apresenta os conceitos necessários para compreender o uso da PSI no desenvolvimento de extensões para o IntelliJ. São abordadas as representações estruturais de código-fonte, o papel das IDEs como ambientes que exigem abstrações avançadas, a importância da extensibilidade por meio de *plugins* e, por fim, a definição da PSI.

2.1. Representação Estrutural de Código-Fonte

O código-fonte de um programa, em sua forma textual, não é suficiente para permitir análises complexas ou transformações automatizadas. Para que ferramentas possam compreender a organização e os significados do código, são necessárias representações estruturais que expressem de maneira hierárquica e formal os elementos que o compõem. Entre essas representações, destacam-se as árvores sintáticas, tradicionalmente classificadas como *Concrete Syntax Tree* (CST) e *Abstract Syntax Tree* (AST) [Aho et al. 2013].

A CST, também chamada de *parse tree*, corresponde a uma árvore que representa a estrutura completa do código, preservando todos os elementos sintáticos. Essa forma detalhada é útil em contextos que exigem a análise literal do texto do programa, como verificações de conformidade com gramáticas ou transformações de baixo nível [Aho et al. 2013]. Já a AST é uma versão mais abstrata, que omite detalhes superficiais da sintaxe e privilegia a estrutura lógica e semântica do código [Cooper and Torczon 2012]. Por exemplo, em uma atribuição, a AST armazena a relação entre variável e expressão, sem necessariamente manter todos os símbolos utilizados para expressar a operação no código original. A título de exemplo, as Figuras 1 e 2 representam, respectivamente, a CST e a AST para a expressão `while(x<10):x=x+1`.

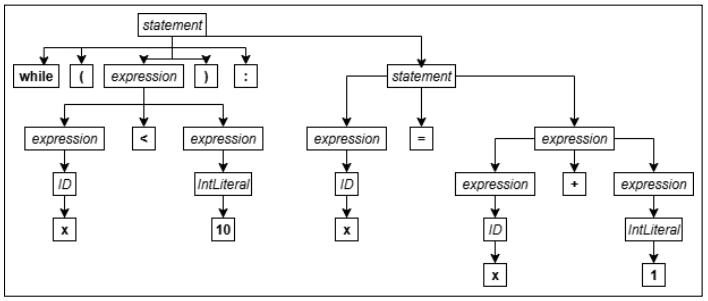

Figura 1. CST

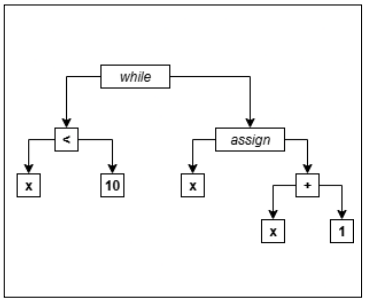

Figura 2. AST

Como pode ser visto nas Figuras 1 e 2, a CST preserva todos os elementos sintáticos da expressão, incluindo parênteses, dois-pontos e símbolos de operação, resultando em uma estrutura mais detalhada. Em contraste, a AST abstrai esses detalhes superficiais, mantendo apenas a estrutura lógica essencial: o laço *while* com sua condição (`x<10`) e o corpo de atribuição (`x=x+1`), criando uma representação mais enxuta.

Essas representações estruturais são fundamentais em diversas atividades de Engenharia de Software. Ferramentas de compilação utilizam ASTs como base para verificação semântica e geração de código intermediário [Mogensen 2009]. Ambientes de desenvolvimento fazem uso dessas estruturas para oferecer recursos de navegação, inspeções estáticas, refatorações automatizadas e geração de sugestões inteligentes [Murphy et al. 2006]. No contexto deste estudo, as árvores sintáticas fornecem o ponto de partida conceitual para compreender a PSI. Embora a PSI não seja equivalente a uma AST tradicional, ela compartilha a mesma motivação: oferecer uma estrutura hierárquica que permita a exploração, análise e transformação do código-fonte.

2.2. Extensões em Ambientes de Programação

Ambientes de desenvolvimento modernos são projetados para atender a uma ampla variedade de linguagens, paradigmas e fluxos de trabalho [Harmanen and Mikkonen 2016]. Dada a diversidade de necessidades dos desenvolvedores, torna-se inviável que uma IDE ofereça de forma nativa todas as funcionalidades desejadas em diferentes contextos [Kurbatova et al. 2021]. Por isso, muitas plataformas de desenvolvimento adotam arquiteturas extensíveis, nas quais recursos adicionais podem ser incorporados por meio de *plugins*. Essa abordagem possibilita que a comunidade de usuários contribua com funcionalidades específicas para as mais distintas necessidades, tais como integração com sistemas de controle de versão, temas gráficos, emuladores de dispositivos, etc.

O modelo baseado em extensões traz benefícios tanto para os desenvolvedores quanto para os fabricantes de IDEs. Para os usuários, a principal vantagem é a possibilidade de personalizar o ambiente de acordo com as necessidades do projeto, sem depender exclusivamente da equipe mantenedora da ferramenta. Já para as plataformas, a extensibilidade cria um ecossistema colaborativo, em que a evolução das funcionalidades ocorre de forma distribuída. Exemplos reconhecidos incluem o Eclipse[1], que consolidou sua relevância por meio de um ecossistema de *plugins*, e o Visual Studio Code[2], que popularizou a adoção de extensões para suportar linguagens e *frameworks*.

2.3. IntelliJ e a *Program Structure Interface*

O IntelliJ IDEA, desenvolvido pela JetBrains, é uma das IDEs mais consolidadas para o desenvolvimento em Java [StackOverflow 2024]. Apresenta um conjunto abrangente de funcionalidades, como refatorações, inspeções de código em tempo real, geração automática de trechos [JetBrains 2025a]. Além disso, a plataforma foi concebida desde suas primeiras versões como uma base extensível, permitindo a criação de *plugins* que expandem suas capacidades para atender a cenários específicos de desenvolvimento.

No núcleo dessa extensibilidade encontra-se a *Program Structure Interface*, uma abstração que representa o código-fonte em uma estrutura hierárquica manipulável. A PSI

[1] `https://marketplace.eclipse.org/listings/category/ide`

[2] `https://code.visualstudio.com/docs/configure/extensions/extension-marketplace`

pode ser entendida como uma camada acima da *Abstract Syntax Tree* do Java, oferecendo não apenas a estrutura sintática do programa, mas também elementos adicionais que facilitam sua manipulação em um ambiente de edição. Dessa forma, ela atua como a ponte entre a representação sintática bruta do código e as operações de alto nível realizadas pela IDE, como navegação, inspeção e transformação [JetBrains 2025f]. A PSI organiza o código em entidades chamadas `PSIElement`, que correspondem a nós individuais da árvore estrutural, como classes, métodos, expressões ou variáveis [JetBrains 2025d]. Esses elementos são agregados em estruturas maiores, sendo o `PSIFile` a unidade de nível superior que representa um arquivo-fonte completo [JetBrains 2025e]. Além disso, a PSI permite explorar essas estruturas por meio de navegação hierárquica, podendo seguir uma abordagem *top-down*, partindo do arquivo até os elementos mais internos, ou *bottom-up*, subindo a partir de um nó específico até seus elementos ancestrais [JetBrains 2025c]. Essa flexibilidade a torna uma ferramenta adequada para implementar análises e manipulações precisas do código dentro de *plugins*.

3. Metodologia

A pesquisa adotou a abordagem de *Design Science Research* (DSR), que tem como objetivo principal a construção de artefatos capazes de solucionar problemas reais [Peffers et al. 2007]. Conforme descrito por [Hevner et al. 2004], a DSR propõe um equilíbrio entre teoria e relevância prática, favorecendo a criação de soluções cientificamente embasadas e, ao mesmo tempo, aplicáveis no ambiente em que são implementadas.

A DSR é estruturado em três ciclos: o ciclo de relevância, que garante que o artefato atenda às demandas do contexto prático; o ciclo de rigor, que assegura que o desenvolvimento esteja fundamentado em conhecimento consolidado e literatura pertinente; e o ciclo de *design*, que envolve a construção e refinamento contínuo do artefato. A interação desses ciclos possibilita que os resultados obtidos sejam simultaneamente úteis para a prática e consistentes do ponto de vista científico [Horita et al. 2018].

Neste estudo, a aplicação da DSR iniciou pelo planejamento e definição dos requisitos dos artefatos, e posterior desenvolvimento dos *plugins*. O trabalho se comprometeu com cada ciclo da DSR da seguinte forma: no **(I) ciclo de relevância**, o problema foi mapeado a partir das lacunas identificadas no uso de abstrações estruturais em IDEs; no **(II) ciclo de rigor**, foram considerados fundamentos teóricos sobre ASTs, CSTs, PSI e extensibilidade de IDEs, garantindo que o desenvolvimento fosse consistente com a literatura; e no **(III) ciclo de *design***, os artefatos foram projetados, implementados e avaliados iterativamente, permitindo ajustes até a obtenção de soluções funcionais.

4. Desenvolvimento dos Artefatos

Nesta seção são apresentados os *plugins* desenvolvidos como prova de conceito, ilustrando a aplicação prática da PSI no IntelliJ por meio de exemplos que exploram diferentes formas de análise e manipulação do código-fonte.

4.1. *Editor Context Info*

O *plugin* foi desenvolvido com o objetivo de exibir informações contextuais sobre a posição atual do cursor no editor. É um exemplo conceitual cujo objetivo é fornecer ao desenvolvedor, detalhes como o nome do arquivo, a linha e a coluna em que o cursor está localizado, bem como os métodos e classes que envolvem o ponto de edição.

A execução do *plugin* ocorre em quatro etapas: **(I)** inicialmente, a partir do objeto `AnActionEvent`, é realizada a recuperação do contexto, obtendo-se a instância do projeto e o editor ativo; **(II)** em seguida, com o uso da API de edição (`CaretModel`), identifica-se o posicionamento do cursor, determinando o deslocamento no texto e a posição lógica (linha e coluna); **(III)** na sequência, ocorre a exploração da PSI, em que o documento é associado a um objeto `PsiFile`, permitindo navegar na estrutura do código e, a partir da posição do cursor, identificar o elemento correspondente (`PsiElement`) e seus ancestrais, como `PsiMethod` e `PsiClass` por meio do `PsiTreeUtil`; **(IV)** por fim, as informações coletadas são organizadas em uma mensagem e exibidas ao usuário em uma janela de diálogo, consolidando a integração entre a posição textual e a estrutura.

```
    PsiElement elementAtCursor = psiFile.findElementAt(offset);
    if (elementAtCursor != null) {
        PsiMethod method = PsiTreeUtil.getParentOfType(elementAtCursor, PsiMethod.class);
        PsiClass psiClass = PsiTreeUtil.getParentOfType(elementAtCursor, PsiClass.class);

        if (method != null) {
            methodName = method.getName();
        }

        if (psiClass != null) {
            className = psiClass.getName();
        }
    }
```

Figura 3. Trecho do código-fonte do `EditorContextInfo`

O fragmento de código-fonte contido na Figura 3 ilustra como a PSI permite navegar hierarquicamente na estrutura do código. Primeiro, `findElementAt(offset)` localiza o elemento PSI exato na posição do cursor. Em seguida, `PsiTreeUtil.getParentOfType()` percorre os ancestrais desse elemento, buscando especificamente por `PsiMethod` e `PsiClass`. Essa navegação ascendente permite identificar o contexto (método e classe) que envolve o ponto de edição, independentemente da profundidade na hierarquia.

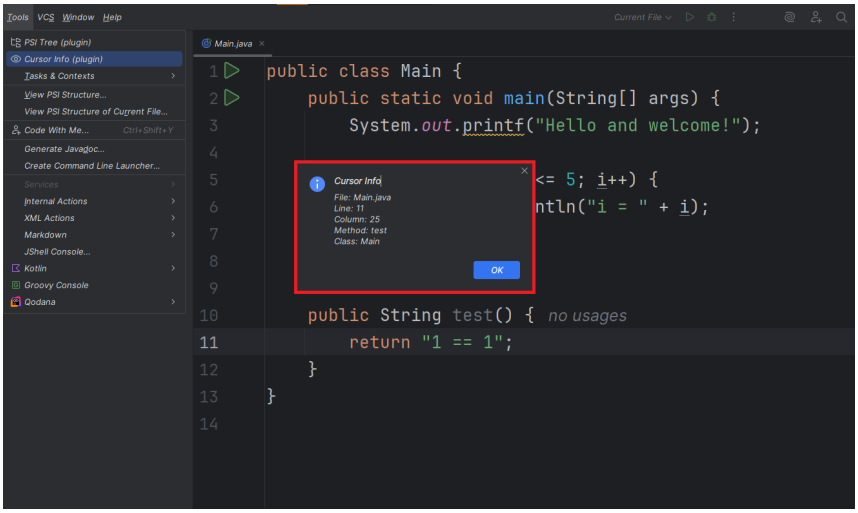

Figura 4. Aplicação prática do `EditorContextInfo`

A Figura 4 demonstra o *plugin Editor Context Info* em funcionamento. O cursor está posicionado na linha 11, coluna 25, dentro do método `test()` da classe `Main`. O diálogo exibe informações contextuais extraídas pela PSI: o arquivo, posição exata do

cursor (linha/coluna) e hierarquia estrutural (método e classe). Isso exemplifica como a PSI correlaciona a posição textual com a estrutura semântica do código.

4.2. PSI *Tree Generator*

O *plugin* foi desenvolvido com o objetivo de gerar uma visualização textual da árvore de elementos PSI de um arquivo em edição. Ele permite que o desenvolvedor compreenda a organização hierárquica do código-fonte conforme representada pela *Program Structure Interface*, exibindo informações como o tipo dos elementos, seus nós sintáticos e trechos de texto associados. Essa visualização pode ser especialmente útil para fins de depuração ou análise, oferecendo uma visão detalhada das estruturas internas.

A execução do *plugin* ocorre em quatro etapas: **(I)** inicialmente, a partir do objeto `AnActionEvent`, é realizada a recuperação do contexto, obtendo-se a instância do projeto e o editor ativo; **(II)** em seguida, o documento é associado a um objeto `PsiFile`, que contém a representação PSI do arquivo; **(III)** a partir desse objeto, é percorrida recursivamente toda a árvore de elementos por meio do método `buildPsiTree`, que constrói uma saída textual com indentação hierárquica e informações como classe do elemento, tipo do nó e texto associado; **(IV)** por fim, a árvore gerada é apresentada em uma janela de diálogo personalizada, permitindo rolagem e navegação pelo conteúdo exibido.

```
1     private void buildPsiTree(PsiElement element, StringBuilder builder, String prefix, boolean isLast) {
2             // Adiciona os conectores
3             builder.append(prefix);
4             builder.append(isLast ? "|_" : "|-");
5
6             builder.append(element.getClass().getSimpleName());
7
8             // ... Implementacao omitida ...
9
10            // Processa os elementos filhos recursivamente
11            PsiElement[] children = element.getChildren();
12            for (int i = 0; i < children.length; i++) {
13                boolean isLastChild = (i == children.length - 1);
14                String childPrefix = prefix + (isLast ? "    " : "|   ");
15                buildPsiTree(children[i], builder, childPrefix, isLastChild);
16            }
17        }
```

Figura 5. Trecho do código-fonte do `PSITree`

O fragmento de código-fonte contido na Figura 5 ilustra como o método `buildPsiTree` implementa um algoritmo recursivo de travessia em profundidade para gerar uma representação textual hierárquica da árvore PSI. A função recebe quatro parâmetros: o elemento PSI atual, um `StringBuilder` para construir a saída, um prefixo de indentação e um booleano indicando se é o último elemento no nível. Inicialmente, adiciona conectores visuais seguidos do nome da classe do elemento via `getClass().getSimpleName()`. A recursão processa todos os elementos filhos através de `element.getChildren()`, calculando dinamicamente o prefixo de indentação para cada nível: adiciona espaços em branco quando o elemento pai é o último ou uma barra vertical com espaços quando há mais elementos no mesmo nível. Essa lógica garante que a visualização reflita corretamente a hierarquia da árvore PSI, criando uma saída legível que permite compreender a organização completa do código-fonte.

A Figura 6 apresenta a saída do *plugin* PSI *Tree Generator* aplicado ao arquivo `Main.java`. A visualização revela a estrutura hierárquica completa do código, desde o elemento raiz `PsiJavaFileImpl` até os componentes mais granulares como tokens

e identificadores. Vale ressaltar que a figura não está mostrando a estrutura completa da árvore que foi gerada, como evidenciado pela barra de rolagem.

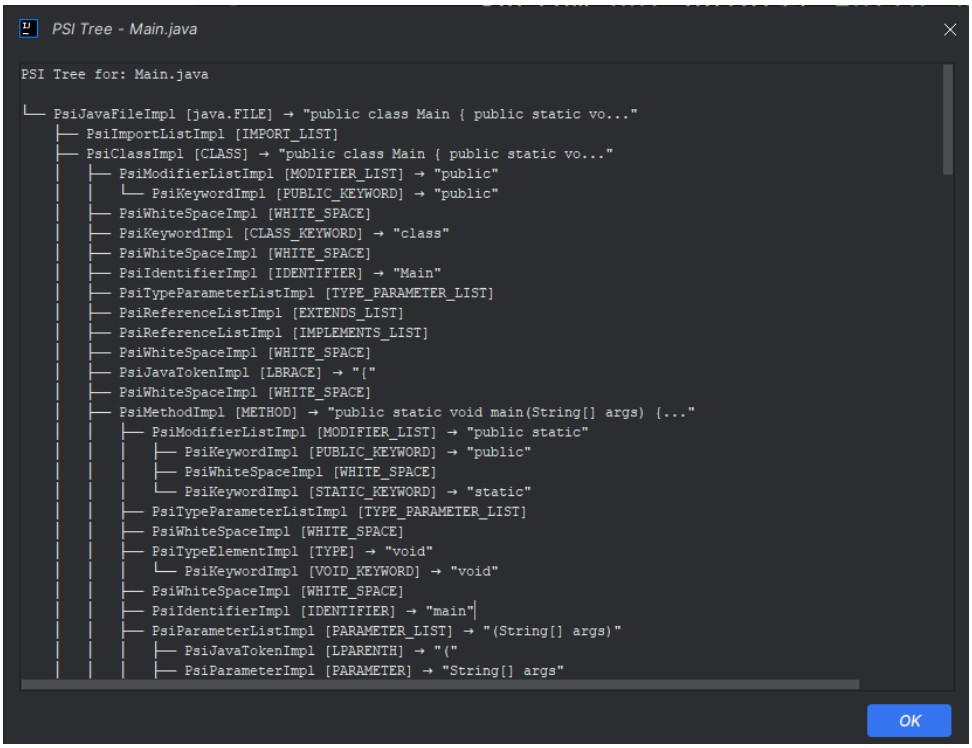

Figura 6. Aplicação prática do `PSITree`

5. Discussão

Os dois *plugins* desenvolvidos demonstraram, de maneira prática, a aplicabilidade da PSI como abstração para manipulação e análise do código-fonte dentro do IntelliJ. O *Editor Context Info* evidenciou a possibilidade de correlacionar a posição do cursor no editor com a hierarquia estrutural do código, extraindo informações relevantes sobre classes e métodos. Já o PSI *Tree Generator* apresentou a capacidade de percorrer e visualizar toda a árvore estrutural de um arquivo, tornando explícitas as relações entre elementos sintáticos e seus respectivos conteúdos. Ambos alcançaram os objetivos propostos, confirmando que a PSI pode ser utilizada tanto em casos pontuais quanto em análises completas.

Ademais, embora a PSI compartilhe semelhanças com a AST, principalmente no que se refere à representação hierárquica do código, o desenvolvimento dos *plugins* revelou que há diferenças relevantes que justificam seu uso específico na IntelliJ IDEA.

A Tabela 1 evidencia diferenças relevantes entre AST e PSI. Enquanto a AST representa a estrutura lógica do código e é tradicionalmente usada em compiladores para análise semântica, a PSI foi projetada para o contexto das IDEs, incorporando metadados e recursos que possibilitam navegação, inspeções e refatorações em tempo real. Essa distinção justifica o uso da PSI como base para o desenvolvimento de *plugins*, pois oferece suporte direto a operações interativas que ampliam a produtividade do programador.

Tabela 1. Comparativo entre AST e PSI

Aspecto	AST	PSI
Origem	Derivada diretamente da gramática da linguagem.	Construída a partir da AST, mas enriquecida com elementos adicionais da IDE.
Detalhamento	Representa a estrutura lógica e semântica do programa, omitindo detalhes supérfluos da sintaxe.	Inclui, além da estrutura lógica, informações úteis ao editor, como referências de navegação e manipulação.
Uso típico	Compiladores e analisadores semânticos.	IDEs, análise estática, refatorações e suporte à navegação de código.
Manipulação	Estritamente ligada ao processo de compilação e interpretação.	Projetada para permitir interatividade em tempo real no ambiente de edição.

6. Considerações Finais

O estudo evidenciou o papel da *Program Structure Interface* como uma abstração fundamental para a construção de funcionalidades avançadas no IntelliJ. A implementação dos *plugins* desenvolvidos demonstrou, de maneira prática, como a PSI possibilita tanto a exploração de informações contextuais quanto a visualização hierárquica detalhada do código, reforçando sua utilidade em apoio ao programador. Apesar dos resultados positivos, algumas limitações devem ser destacadas. O escopo do trabalho restringiu-se a dois artefatos conceituais, o que limita a generalização. Além disso, não foram realizadas avaliações quantitativas de desempenho ou usabilidade, o que poderia fornecer uma visão mais robusta sobre a eficácia das soluções.

Como trabalhos futuros, sugere-se a criação de *plugins* mais complexos, a ampliação dos testes em projetos reais de maior escala e a comparação com outras abordagens de representação estrutural em diferentes ambientes de programação. Um exemplo concreto seria o desenvolvimento de uma ferramenta de model-based programming para Java no IntelliJ que possibilitasse tanto a representação gráfica do código-fonte quanto o rastreamento de mudanças realizadas no modelo, tendo como indispensável o subsídio do modelo de dados do código-fonte provido pela PSI.

7. Disponibilidade de Dados

Nos comprometemos a promover a transparência e a reprodutibilidade na pesquisa. Alinhados com esse princípio, disponibilizamos abertamente o código-fonte dos *plugins* desenvolvidos em nosso estudo no repositório Zenodo em `https://doi.org/10.5281/zenodo.17220255`.

Referências

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2013). *Compilers: Pearson new international edition*. Pearson Education, 2 edition.

Cooper, K. D. and Torczon, L. (2012). Overview of compilation. In *Engineering a Compiler*. Elsevier.

Fowler, M. (2019). *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2 edition.

Golubev, Y., Kurbatova, Z., AlOmar, E. A., Bryksin, T., and Mkaouer, M. W. (2021). One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on ESEC/FSE 2021*. ACM.

Harmanen, J. and Mikkonen, T. (2016). *On Polyglot Programming in the Web*. IGI Global.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*.

Horita, F., Graciano Neto, V., and dos Santos, R. (2018). *Design Science Research em Sistemas de Informação e Engenharia de Software: Conceitos, Aplicações e Trabalhos Futuros*.

JetBrains (2025a). Intellij IDEA. `https://www.jetbrains.com/pt-br/idea/`.

JetBrains (2025b). Intellij Plugins. `https://plugins.jetbrains.com/docs/intellij/developing-plugins.html`.

JetBrains (2025c). Navigating the PSI. `https://plugins.jetbrains.com/docs/intellij/navigating-psi.html`.

JetBrains (2025d). PSI Elements. `https://plugins.jetbrains.com/docs/intellij/psi-elements.html`.

JetBrains (2025e). PSI Files. `https://plugins.jetbrains.com/docs/intellij/psi-files.html`.

JetBrains (2025f). What is the PSI? `https://plugins.jetbrains.com/docs/intellij/psi.html`.

Kurbatova, Z., Golubev, Y., Kovalenko, V., and Bryksin, T. (2021). The intellij platform: A framework for building plugins and mining software data. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE.

Leite, L., Rocha, C., Kon, F., Milojicic, D., and Meirelles, P. (2019). A survey of devops concepts and challenges. *ACM Computing Surveys*.

Mogensen, T. Æ. (2009). *Basics of compiler design*.

Murphy, G., Kersten, M., and Findlater, L. (2006). How are java software developers using the eclipse ide? *IEEE Software*.

Peffers, K., Tuunanen, T., Rothenberger, M., and Chatterjee, S. (2007). A design science research methodology for information systems research. *J. Manage. Inf. Syst.*

StackOverflow (2024). Most popular technologies report. `https://survey.stackoverflow.co/2024/technology#most-popular-technologies-new-collab-tools`.