Writable PSI Generator for a Multi-Language IDE Platform

Alexander Bozhnyuk
Saint-Petersburg State University
Mathematics & Mechanics Faculty
Saint Petersburg, Russia
bozhnyuks@mail.ru

Alexander Zakharov
Tula State University
The Faculty of Cybernetics
Tula, Russia
lynxsm@gmail.com

Nikolai Tropin
Saint-Petersburg State University
Mathematics & Mechanics Faculty
Saint Petersburg, Russia
niktrop@yandex.ru

Mikhail Volkov
Saint-Petersburg State University
The Faculty of Physics
Saint Petersburg, Russia
mvvolkov@mail.ru

***Abstract*— The Program Structure Interface (PSI) is a special data structure and corresponding API used in IDEs to support code navigation and transformation features. In this paper, an approach for generation of a writable PSI basing on language syntax construct types is proposed (Writable PSI Generation). The approach is developed for a multi-language platform of a large telecommunications company. Refactoring and Quick Fix features are implemented using on the proposed generator for two IDEs: a Python IDE and a Java IDE.**

Keywords—Integrated Development Environment (IDE), Development Services, Program Structure Interface (PSI), Application Program Interface (API), Refactoring, Quick Fix

I. INTRODUCTION

An Integrated Development Environment (IDE) is an essential tool for any programmer. Some of the most well-known and widely used IDEs are JetBrains IntelliJ IDEA and Microsoft Visual Studio, which offer a large number of features to develop high-quality software.

One of the most important tasks of the IDE is to provide developers the ability to quickly and correctly modify the source code. To achieve this, IDEs offer such features as refactoring and quick fixes. Refactoring makes it possible to restructure code while preserving its semantics, for example, to rename a class, method, and attribute, extract selected code into a method, and so on. Quick fixes, at the request of the developer, eliminate a drawback of a code fragment. An example of a quick fix is if statement simplification.

These features work with the structure of the program by analyzing and reorganizing it. The conventional way of representing a program internally is an Abstract Syntax Tree (AST) that is generated via parsing [1]. However, IDEs often need to work with additional semantic information (for example, to determine the declaration of a method or attribute by its occurrence), which would also be convenient to store in the tree. Therefore, the IDE builds another tree on top of the AST, which gives external clients (IDE features) access to such information about the program. In IntelliJ IDEA, such a tree is called Program Structure Interface (PSI) [2], and this name is used in this paper. Thus, PSI stores additional information and provides clients with a rich API, and AST is an implementation detail.

For convenience, in PSI each of its nodes has its own type according to the syntax structure of the language that this node represents, within the syntax construct type system of the programming language in which this tree is created. For example, in the context of the Java language, each node is defined by its own class (PsiFunction, PsiClass, etc.). At the same time, different IDEs implement various approaches to building such a data structure and methods of interacting with it [3, 4].

IDE features, after manipulating PSI, transfer the changes to the source code so that they become visible to the developer. To do this, text changes are generated based on the changes in the tree, which are then applied to the code.

As mentioned above, PSI is typed. This is convenient, but the types in such a tree must be accessed somehow in the source code. If the IDE, for example, is developed in Java, then one will need to create a large number of interfaces and classes for that purpose. This process is very time-consuming due to the large number of types, and therefore highly error-prone. For this reason, it is desirable to use generation, which is based on a pre-created specification of syntax construct types of the programming language.

A large telecommunications company is developing a multi-language platform for effectively creating IDEs for different programming languages. Two IDEs (for Java and Python) are being created at the moment.

The platform requires a unified system for managing the source code structure. Each specific IDE requires its own PSI tree and tools for manipulating it, as well as a system for displaying changes in the code. However, the principles of generating PSI access interfaces based on programming language construct types are universal and can be implemented within the platform and used in various IDEs.

The main contributions of the paper are as follows.

- Design of the Writable PSI Generator architecture: a mechanism for generating classes and interfaces for accessing the PSI tree, as well as a single mechanism for distributing text changes.
- Implementation of the Writable PSI Generator:
  - A component for generation of the necessary Java interfaces and classes for the PSI tree modification system based on JSON specification.

- A component for modifying the tree consisting of a persistent tree and a Rewriter, and a mechanism for obtaining text changes (the GumTree algorithm) [5].

- The Writable PSI Generator was successfully tested in the Java and Python IDEs in the implementation of a number of refactoring services and quick fixes.

The remainder of this paper is organized as follows: in Section II, we present functional requirements and the architecture of the Writable PSI Generator. Section III presents system implementation details. Further, we discuss the convenience of the Writable PSI Generator and show the success of reuse in Section IV. Finally, Section V presents related work.

II. Architecture

The functional requirements of the Writable PSI Generator are the following.

- The system should allow for generating Java interfaces and classes for working with PSI (the main language within the multi-language platform used for developing various IDEs is Java).
- The system should allow for modifying the tree for the needs of refactorings and quick fixes.
- It should be possible to get text changes for the source code document.
- It is necessary to ensure that the system can be reused for various IDEs developed within a multi-language platform.

The Writable PSI Generator consists of two subsystems: the subsystem for transforming the PSI and obtaining text changes, and the subsystem for generating interfaces and classes access to the PSI tree.

Fig. 1 shows an UML component diagram describing the subsystem for transforming the PSI and obtaining text changes in the Writable PSI Generator.

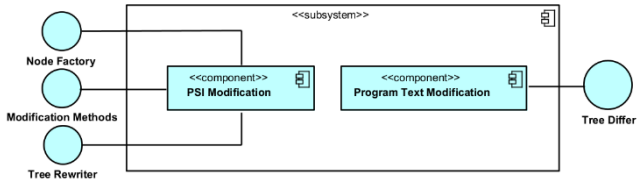

Fig. 1. Subsystem for transforming the PSI and obtaining text changes

The PSI Modification component provides external clients with different ways to modify and build new PSI nodes:

- Node Factory is a factory that provides methods for both constructing PSI nodes from other nodes and creating them from a string. This factory is generated, but at the same time it is possible for it to "manually" add additional methods.
- Modification Methods are generated methods that each interface and class contain for modifying the attributes of the syntax construct. These methods allow the client to create a new version of the node, replacing existing children.
- Tree Rewriter is an entity that allows the client to create a new copy of PSI by replacing or removing some nodes. Implements the Builder design pattern.

The Program Text Modification component provides external services with the ability to receive text changes to a document after PSI transformations. The client is provided with a Tree Differ, which, after receiving two trees, finds differences between them and creates the sequence of text changes that the client can apply to the source code document.

Fig. 2 shows a UML sequence diagram which describes the main scenario of using the subsystem for transforming the PSI and obtaining text changes.

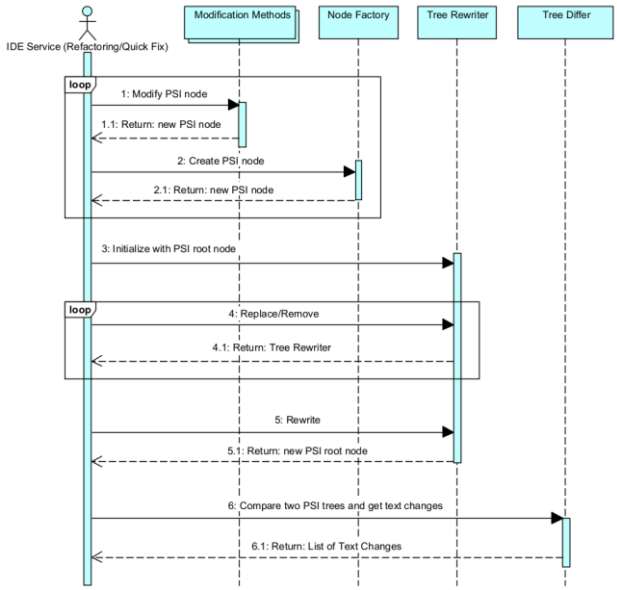

Fig. 2. The main scenario of using the subsystem for transforming the PSI and obtaining text changes

It includes the following steps:

- Modification of tree nodes (1) or construction of new nodes using the factory (2). As a result, new tree nodes are available for the external service.
- Modification of the entire PSI is performed using the Tree Rewriter interface, as a result of which a new, modified copy of the PSI is created. Firstly, the feature initializes this interface with the root of the new PSI (3). Secondly, then through the replace/remove methods it indicates which transformations need to be performed (4). Finally, using the Rewrite method (5), the modification process is activated, and as a result, the feature receives a new PSI.
- Obtaining a sequence of text changes using the Tree Differ interface, which takes the roots of the old and new PSI as input, compares them and creates a specification of text changes (6).

Fig. 3. describes the subsystem for generating interfaces and classes for PSI tree access. It consists of the following components:

- Specification Processor is responsible for processing and validating the pre-written developer specification of the syntax construct types of the programming language for which the PSI is being built.
- Scheme Manager stores knowledge about the schemes for generating Java interfaces and classes created based on type information from Specification Processor: which interfaces are implemented, the order of children during generation, and so on. Scheme Manager implements the Singleton design pattern.
- Types Manager stores knowledge about the semantic of syntax construct types: types of children, properties, etc. Similarly to Scheme Manager, it implements the Singleton design pattern.
- Generation is the main component that contains everything related to PSI generation. It provides the Generator interface, which is responsible for generating a specific file.

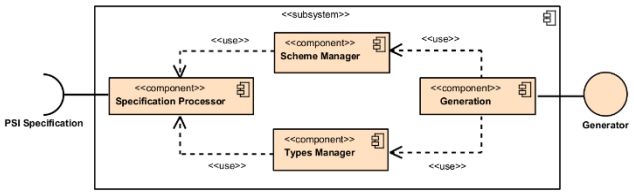

Fig. 3 The subsystem for generating interfaces and classes for PSI tree access

As a result of using the Writable PSI Generator, a user who wants to generate PSI for their IDE only needs to write a specification of types of syntax constructs for the corresponding programming language and run the generator. If necessary, Writable PSI Generator can be extended to take into account the specifics of a particular language.

III. Implementation Details

This section discusses the features and implementation details of the components described in Section II.

A. *PSI Modification Component*

As mentioned in Section II, the PSI Modification component is responsible for modifying and creating new PSI nodes. It also provides the functionality to completely rewrite the entire file tree.

PSI is a Lossless Syntax Tree (LST) [3], i.e., it has the following features.

- It stores information about whitespaces and comments in special nodes called Trivia.
- Every PSI node stores its source text position and length.

Fig. 4 shows an example of a Lossless Syntax Tree for a simple Java return statement.

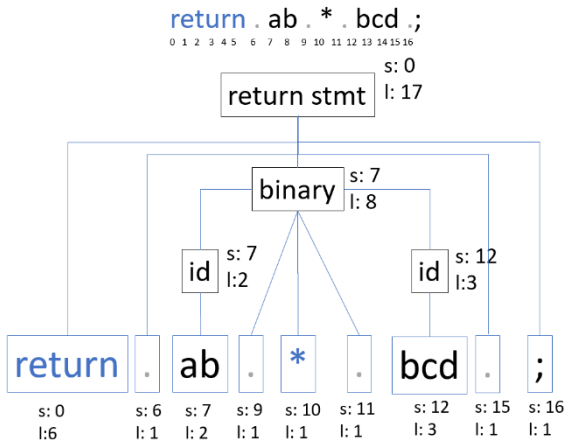

Fig. 4. Lossless Syntax Tree example

We chose Persistent Tree as the main approach for building the PSI and its modification system. Here, persistence means that when a data structure is modified, a new version of this data structure is returned. In addition, the unchanged parts of the data structure are reused. This approach provides the following benefits.

- Thread-safety via PSI immutability, as it eliminates the need for synchronization. In the IntelliJ Platform, for example, it is necessary to use Read and Write Action to interact with the PSI because of tree mutability [10].
- Fixed offsets and lengths of nodes in the tree. Immutability makes it possible not to be concerned about updating node offset in the text, as it will be correct after recreating the node.
- Secure manipulation of semantic information via separating the stages of tree modification. The user clearly knows when the semantic information is relevant.

In constructing this data structure, we opted for the Red-Green-Trees method from Microsoft Roslyn [4]. This approach results in PSI constructed as a combination of two trees.

- The Green tree is an immutable untyped tree built during parsing. Its nodes (green) store information about node length in text, type, etc. They also store references to their children, but not to their parents. This tree is an implementation detail, and it is kept hidden from clients.
- The Red tree is an immutable typed tree, which is built lazily on demand from top to bottom. This is the PSI that the client works with. The nodes of this tree (red) reference the corresponding green nodes. Each red node stores an offset in the text document and a reference to its parent.

This approach to PSI construction resulted in a correctly working persistent data structure. The two trees are needed to provide the ability to iterate over the parents and children of PSI nodes.

Figure 5 shows a simple example of this approach.

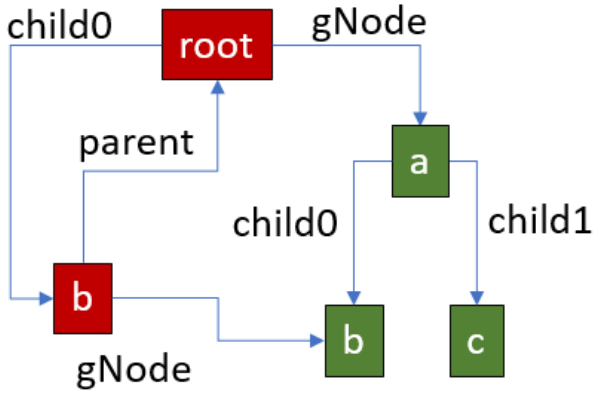

Fig. 5. An example of the Red Nodes and Green Nodes approach

In order to reduce the number of errors when working with the developed subsystem, we proposed an identifier system where each green node is assigned an identifier. This identifier is transferred to the new version when modifying and creating a new node. Modification methods and factory methods take these identifiers into account, which made it possible to build a more convenient API and support more PSI Modification Component usage scenarios.

As mentioned earlier, PSI stores Trivia nodes with whitespaces and comments. Microsoft Roslyn maintains the invariant that a node is Trivia if and only if it is a child of a token. This invariant is convenient for compiler system development, because it eliminates the problem of space and comment placement, leaving it as the responsibility of the client.

This approach was not applicable in the context of our IDE platform due to complicated API and difficulties in developing external services, and therefore Trivia nodes were placed in a more classical way — at the token level. The PSI Modification Component is responsible for whitespace normalization itself during node modification. Fig. 6 (left) illustrates Trivia node placement in Microsoft Roslyn, while Fig. 6 (right) shows the same for the Writable PSI Generator.

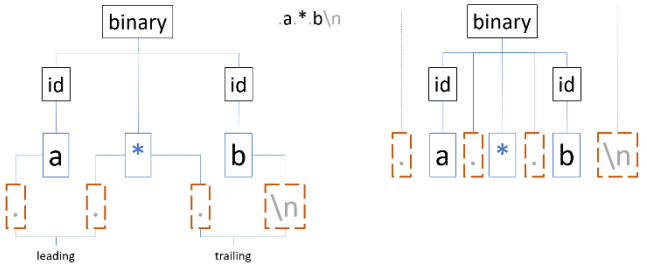

Fig. 6. Trivia nodes in Microsoft Roslyn and Writable PSI Generator

Modification Methods and Node Factory, which were mentioned in Section II, are based on a common system for green node manipulation. Every Node Factory method makes it possible to build a new green node from other existing green nodes. A new red node is created based on the new green node. Every modification method uses the Node Factory method to build a new node based on the existing children. These methods are uniform and easy to generate. Furthermore, they are built based on the information about the syntax constructs of the language described in the specification, and thus produce only syntax-correct nodes.

The considered component also provides Tree Rewriter, an object that allows to replace or delete nodes in the PSI of the whole file. The replace and rewrite methods let Rewriter accumulate information about what changes should be applied to the tree. This is done by filling in the replace map and remove list, which store the data about the accumulated changes. The rewrite method activates PSI traversal, during which Rewriter replaces or removes nodes. This traversal is a Preorder Traversal, where the node itself is processed first, followed by its children from left to right. Rewriter takes into account the node offsets in source text, and therefore it does not have to traverse the whole tree. Instead, it only traverses the parts which have something to transform.

The result of the traversal is a new PSI. Rewriter takes into account the syntax structure of language constructions described in the specification, and does not produce a PSI with syntax errors.

As a result, the PSI Modification component meets the functional requirements described in Section II, and achieves the following.

- Thread-safety.
- Syntax correctness after PSI transformation.
- Generatable API for modifying PSI nodes and producing new ones.
- Possibility to safely transform the PSI of an entire file.

B. Program Text Modification Component

As mentioned in Section II, the Program Text Modification component is responsible for creating the shortest sequence of text changes that can be applied to the source document.

Three types of text changes are implemented:

- text insertion;
- text deletion;
- text replacement.

Each text change has the following structure.

- Position in the document at which the change starts.
- Position in the document at which the change ends. In the case of insertion, it is equal to the start position.
- The text to replace the fragment in the document. In the case of deletion, this string is empty.

This structure of text changes is due to the specifics of the IDE platform, for which the Writable PSI Generator was implemented.

In implementing the component, we decided to follow the GumTree approach [17], which produces text changes in two stages.

- First, it establishes the mappings between the nodes of the initial and final trees

- Then, it analyzes these mappings and constructs a sequence of text changes based on the analysis.

GumTree made it possible to implement Program Text Modification, which generates the text sequence accurately and quickly. However, this approach required adaptation to the specifics of the developed IDE.

- GumTree allows to generate changes to move subtrees, which are not supported by the IDE. Therefore, these changes have been replaced by appropriate deletions and insertions.
- Text changes in the IDE platform are not applied sequentially — they are applied simultaneously. The approach has been adapted so that the created text changes meet the requirements of the platform. For example, multiple additions in a sequence in the same area are merged into a single change.

Such corrections allowed not only to adapt the approach to the requirements of the developed platform, but also to make them more convenient and less confusing, which was important when debugging the developed external services.

The implemented component is designed so that it can be applied to PSIs of different languages. The component itself has no knowledge of which programming language's trees it is analyzing.

C. Generation Subsystem

As it was mentioned in Section II, the Generation Subsystem provides the ability to generate interfaces and classes to work with the PSI Modification component.

This subsystem is based on a given specification. The JSON format was chosen since it is widespread and has convenient processing and generation tools.

The specification contains the information necessary both for the operation of the generator and for the correct functioning of the entire modification subsystem.

- Definition of PSI node types according to programming language syntax. It describes what kind of children the PSI node can have according to the grammar of the language. Modification methods and factory methods are generated based on this information.
- Additional information for the generator. For example, it can specify if the class generated for a given type should be abstract, or if a factory method should be generated for a particular PSI node type, among others.

The specification is processed in several steps.

- Parsing and validation of the specification file
- Initialization of the Types Manager component based on the result of the first step
- Initialization of Scheme Manager component based on the result of the first step

During these steps, the specification file is validated to prevent unexpected system behavior. The following checks are performed:

- Check the presence of all mandatory attributes
- Check all attribute types
- Check presence of unnecessary attributes in JSON objects
- Check correctness of attribute values

The initialized Types Manager and Scheme Manager are objects that implement the Singleton pattern. They are available to both the generator and the PSI Modification component.

The generator is based on a Java StringBuilder, which builds a string that is the content of the generated file based on information from Types Manager and Scheme Manager. This string contains the package name, imports, fields, constructors, methods, etc., and it is written to the desired file. Such generation approach appeared to be the most suitable in the context of the IDE platform due to its simplicity and sufficient flexibility.

The described generation approach addressed another problem as well. Typically, generators produce files that prohibit manual code additions, because repeated generation of additional text is overwritten. However, the generator that we developed can create areas where code is not overwritten and it is possible to add new logic. This is done as follows.

- The generator checks if the file exists on disk.
- If the file does not exist, it is generated. The code is partitioned, leading to the division of the file into areas. Within one group of these areas, the code cannot be re-generated (e.g., the zone of generated methods, the zone of generated fields, etc.). The user can write code in these areas, and they are not re-generated.
- If the file exists, the generator recognizes via special area markers where the re-generation should be performed. The re-generated areas are replaced by new ones, the rest remain unchanged.
- The updated file is obtained by concatenating the contents of the generated and non-generated areas.

Thus, developers are able to implement additional logic in the generated interfaces and classes. Fig. 7 illustrates how a Java interface is derived from the type specification of syntactic constructs. This figure also showcases the division of code into areas where generation does or does not take place.

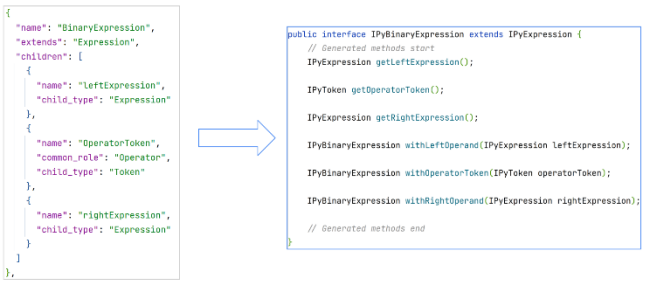

Fig. 7. Example of interface generation based on a JSON specification

IV. Use Cases

This section shows how versatile and convenient the Writable PSI Generator is. The Java IDE and Python IDE are a software product line developed on the basis of a multi-language platform and its reusable assets [6]. The Writable PSI Generator presented in the paper is one of the reusable assets of the platform.

During the usage of the system by different products of the product line, it was improved: errors were corrected and new features were added. In terms of paper [7], this process is called improvement of reusable assets.

Specifications of syntax construct types for Python and Java languages were created, and based on these specifications, interfaces and classes for Python/Java PSI and other auxiliary code were generated. As a result, 21/7 improvements and bug fixes were made to the Writable PSI Generator in response to requests from the Python/Java teams.

It can be seen that the number of requests for such improvements when using the Writable PSI Generator decreased from product to product, indicating successful reuse of the asset.

Using the system within the Python IDE. Based on Writable PSI Generator for Python, the following features were implemented for the Python IDE.

- Rearrange Code is a refactoring that rearranges program constructs in source code. For example, this feature enables the developer to quickly move selected functions and classes through the source code, and reorder function arguments. It can also move functions out of classes into the external scope if the function is either at the very top or the very bottom of the class.
- Introduce Variable is a refactoring which lets the developer define a new variable for a selected expression, to which it will be assigned.
- MinMax If is a quick fix that allows the developer to turn a construct of the form *if a<b: return a else return b* into *return min (a,b).* There is no such feature in PyCharm at the moment.
- Annotated Assignment is a quick fix that allows the user to remove type annotation in case of chain assignment. (For example, a: int = b = d = 3 turns into a = b = d = 3). Python does not allow for type annotations in case of chain assignment: this is a syntax error. However, at the time of development of the Writable PSI Generator, even though PyCharm indicated an error in this case, it did not offer a quick fix.

Using the system within the Java IDE. Using the Writable PSI Generator for Java, the following features were implemented in the Java IDE.

- Rename Method is a refactoring that allows the developer to rename a class method and all its uses within the project.
- Remove Useless Statement is a quick fix that removes a useless construct in the source code (for example, an empty if statement).
- Simplify Trivial If is a quick fix that replaces an if statement with a return of *true* or *false* depending on a condition with a return with a check of this condition.

V. Related Work

PSI was first introduced in [2] for describing the syntax and semantic information of the developed program in IDE. However, the authors presented only general ideas regarding operation with PSI, without considering major non-trivial tasks associated with the PSI, such as tree modification and program text changing.

A. PSI Modification

Study [9] outlines the problem of refactoring service development and describes approaches to building a tree that is more convenient for IDEs. It highlights that in the IDE context the tree should store spaces and comments, and it should also be able to store positions and lengths of nodes in the text. Consequently, such a tree should be a Lossless Syntax Tree (LST), i.e. a tree which can be fully mapped to the original source code. However, this paper presents only a general view of the problem.

Paper [3] reviews different approaches to PSI design suitable for code refactoring services. It discusses two main approaches: Mutable Tree and Immutable Tree.

Mutable Tree is an approach in which tree nodes can be easily deleted, added or changed. It is quite appealing due to its simple implementation and convenient API, and therefore it is used in tools like IntelliJ Platform [10], Smalltalk Refactoring Browser [11], and CRefactory [12]. However, this approach has many disadvantages, such as problems with updating node offsets and lengths, and the need for synchronization in multi-threaded code.

Immutable Tree is an approach in which the tree cannot be altered once it is created. Paper [3] highlighted two approaches to designing a modification process on such a data structure: Rewriter and Persistent Tree. *Rewriter* is an approach in which all transformations over the tree are delegated to a separate object called Rewriter. The tree itself is not writable. This approach is employed in Eclipse Java Development Tools (JDT) and C/C++ Development Tools (CDT), addressing many of Mutable Tree problems, such as the lack of thread-safety. However, it does not provide an ability to interact with intermediate and final versions of trees during the modification process. *Persistent Tree* is an approach which allows clients to execute transformation actions on the tree. However, with every such operation, they receive an updated version of the tree, reflecting the applied transformations. This approach is used in the Microsoft Roslyn compiler written in C#. Its creators describe [4] its implementation via Red-Green Trees, as described above. This method of PSI construction offers an API through the red tree and hides implementation details in the inner green tree. This approach has all the benefits of an immutable tree, but also provides a more convenient way of interacting with the tree to transform it. A notable disadvantage of this approach is the difficulty of creating a convenient API for clients, which is due to the non-trivial organization of the data structure of Persistent Tree.

B. Program Text Changing

After performing transformations on PSI, it is necessary to transfer the changes to the source document (the program's source code) in order to make them visible to the IDE user (developer). In this case, a large number of fine-grained

program changes can lead to performance issues. This problem is known as the problem of obtaining the shortest sequence of text changes that can be applied to the source document. It reduces to the Tree Differencing problem, which has proven itself to be a long-term research topic.

A set of approaches for Tree Differencing with retrieving text changes for adding, deleting and updating nodes in PSI is described in [13]. The RTED algorithm [14] stands out from this set, but its asymptotic performance is insufficient to meet the requirements of our IDE.

Further work tries not only to improve the asymptotic performance, but also determine the moves of subtrees. This is important because many refactoring services are often reduced to this type of tree operations (e.g., the Rearrange Code refactoring). Paper [15] proposes an algorithm for tree differencing of LaTeX trees. It is better compared to previous approaches, and has good asymptotic behavior, but it also has a significant limitation: the algorithm operates on trees that have a large amount of text in the leaves, which is not true for IDEs.

ChangeDistiller [16] improves on the ideas proposed earlier and makes the approach from [15] more suitable for Abstract Syntax Trees (ASTs). While it improves the asymptotic behavior, it still does not address the aforementioned limitation.

However, this limitation is solved by the GumTree algorithm described in [17]. It has suitable asymptotic performance for the needs of our IDE and is the most suitable for PSI differencing. Moreover, it generates a reasonably accurate sequence of textual changes, which also includes operations for moving subtrees. The disadvantage of this approach is that it can generate confusing textual changes, as discussed in [18]. This can be important when debugging an external service.

Paper [19] attempts to fix this problem by providing improvements for GumTree, which increases the accuracy of textual changes. However, this approach shows the best results only with Java code, and, consequently, is not well suited for a multilanguage platform.

Conclusion

The use of the Writable PSI Generator in IDE development projects for Python and Java has significantly improved the efficiency of PSI development by generating a significant amount of code and reducing the number of errors. Positive feedback has been received from the developers.

It should also be noted that the first product that used the Writable PSI Generator was the Python IDE, and the number of change requests within this implementation is larger than that for the next one. This suggests that improvement of reusable assets took place, which corresponds to the statements made in [7].

As a further direction of our work, we can specify the replacement of JSON for describing the types of programming language syntax constructs with a grammar-like language, for example, EBNF [8].

References

[1] Alfred V. Aho, Ravi. Sethi, Jeffrey D. Ullman. "Compilers: Principles, Techniques, and Tools" 1986, pp. 69-70

[2] Z. Kurbatova, Y. Golubev, V. Kovalenko and T. Bryksin, "The IntelliJ Platform: A Framework for Building Plugins and Mining Software Data," *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, Melbourne, Australia, 2021, pp. 14-17, doi: 10.1109/ASEW52652.2021.00016.

[3] Jeffrey L. Overbey. 2013. Immutable source-mapped abstract syntax tree: a design pattern for refactoring engine APIs. In Proceedings of the 20th Conference on Pattern Languages of Programs (PLoP '13). The Hillside Group, USA, Article 7, 1–8.

[4] Lippert E. Fabulous adventures in coding. Blog. https://ericlippert.com/2012/06/08/red-green-trees/

[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14). Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[6] A Framework for Software Product Line Practice, version 5.0, Software Engineering Institute, 2012, https://resources.sei.cmu.edu/

[7] Popova T.N., Koznov D.V., Tinova A.A., Romanovskij K.YU. Evolyuciya obshchih aktivov v semejstve sredstv reinzhiniringa programmnogo obespecheniya // Sistemnoe programmirovanie, 1 (2005), 184-198 (in Russian).

[8] Pattis, Richard E. EBNF: A Notation to Describe Syntax. ICS.UCI.edu. University of California, Irvine.

[9] Retaining comments when refactoring code / Peter Sommerlad, Guido Zgraggen, Thomas Corbat, Lukas Felber. — 2008. — 01. — P. 653–662.

[10] IntelliJ Platform SDK — Modifying the PSI. — URL: https://plugins.jetbrains.com/docs/intellij/modifying-psi.html.

[11] Roberts Don, Brant John, Johnson Ralph. A Refactoring Tool for Smalltalk. // TAPOS. — 1997. — 01. — Vol. 3. — P. 253–263.

[12] Garrido Alejandra. Program Refactoring in the Presence of Preprocessor Directives : Ph. D. thesis / Alejandra Garrido. — USA : University of Illinois at Urbana-Champaign, 2005. — AAI3199001.

[13] Bille Philip. A survey on tree edit distance and related problems // Theoretical Computer Science. — 2005. — Vol. 337, no. 1. — P. 217–239. — URL: https://www.sciencedirect.com/science/article/pii/S0304397505000174.

[14] Pawlik Mateusz, Augsten Nikolaus. RTED: A Robust Algorithm for the Tree Edit Distance. — 2011. — 1201.0230.

[15] Change Detection in Hierarchically Structured Information / Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, Jennifer Widom // Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data. — SIGMOD '96. — New York, NY, USA : Association for Computing Machinery, 1996. — P. 493–504. —URL: https://doi.org/10.1145/233269.233366.

[16] Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction / Beat Fluri, Michael Wursch, Martin PInzger, Harald Gall // IEEE Transactions on Software Engineering. — 2007. – – Vol. 33, no. 11. — P. 725–743.

[17] Fine-Grained and Accurate Source Code Differencing / Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc et al. // Proceedings of the 29th 66 ACM/IEEE International Conference on Automated Software Engineering. — ASE '14. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 313–324. — URL: https://doi.org/10.1145/2642937.2642982.

[18] Guillermo de la Torre, Romain Robbes, and Alexandre Bergel. 2018. Imprecisions diagnostic in source code deltas. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18). Association for Computing Machinery, New York, NY, USA, 492–502. https://doi.org/10.1145/3196398.3196404

[19] J. Matsumoto, Y. Higo and S. Kusumoto, "Beyond GumTree: A Hybrid Approach to Generate Edit Scripts," 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 2019, pp. 550-554, doi: 10.1109/MSR.2019.00082.