Improving the integrated development environment of a legacy software platform

JELLE HULTER, University of Twente, The Netherlands

Legacy software platforms often have software development tools like integrated development environments (IDEs) which are outdated and do not have modern features which are provided to more modern programming languages. Useful and modern tool support assist the software developer in their daily software developing tasks, which help them save time and improve their software development efficiency and effectiveness. In this research paper, we have taken a look at the Progress OpenEdge development platform and developed a language plugin for the IntelliJ IDEA [21] using their IntelliJ Platform SDK [19]. We have then tested whether the plugin can be used by software developers of the Progress OpenEdge platform by checking whether the plugin is compatible with a open-source software repository.

Additional Key Words and Phrases: Progress OpenEdge, Integrated Development Environment, Code Completion, Syntax Highlighting, Code Documentation

1 INTRODUCTION

As the Progress Software Corporation describes on their website, "Progress OpenEdge is a complete development platform for building dynamic multi-language applications for secure deployment across any platform, any mobile device, and any Cloud." [30] Software for this development platform is written in the Progress OpenEdge Advanced Business Language (ABL). This software is written using the Progress Developer Studio for OpenEdge (PDS) [32], which is an Eclipse [8] based integrated development environment provided by Progress. However, the PDS contains some flaws which negatively impact the development experience of a Progress OpenEdge ABL developer. Below, three of these flaws have been identified.

The code completion in the Progress Developer Studio is quite unintuitive. In the Progress ABL programming language, there is a built-in variable type called a "widget handle", or "handle" for short. A handle points to an object in memory and it allows the developer to interact with this object, just like in most object oriented programming languages. [31] This handler can contain a few dozen of different "subtypes". For example, different handle subtypes are temp-tables (temporary in-memory database tables), datasets (collection of such temp-tables), data-sources (link between temp-tables and actual tables in the database), buffers (spot in memory referencing a record in a (temp-)table, allowing reading and modification of columns in that record), pointers to graphical user interface (GUI) related objects, and many more. The code completion tool in PDS [29] has a feature called "context-filtered autocomplete list", which shows a list of methods and attributes which can be applied to the given object type. However, in the context of handles, this does not quite assist the developer because the syntax autocompletion tool shows all possible attributes and methods for all handle types, resulting in a list of more than 1000 suggested keywords. However, often only a small list of a few dozen methods/attributes are relevant and applicable for a given subtype. This makes it difficult for the developer to actually use the code completion tool for finding the right method or attribute to be used for a handle typed variable.

Also, the syntax highlighting in PDS is sometimes incorrect, which can negatively impacts the readability of the code. It seems like the syntax highlighter gives certain tokens certain colors based upon a regular expression, without using the context of the token like you would expect it to do. For example, it is allowed to use certain keywords as a variable name, where this variable name is then highlighted as if it were a keyword. Also, method comments are highlighted the same way as in-line comments which makes them harder to read in my opinion.

Finally, there is no integration with the documentation in the IDE. Progress has published a specification for writing code documentation called ABLDoc. It is syntactically quite similar to Javadoc [26], but it is not integrated within the PDS in any way: it does not show this documentation when the PDS gives code completion suggestions. This negatively impacts the development experience, because it makes it more difficult to know what a certain method does.

1.1 Platform history

For this section, an article from the Finland Progress User Group has been used, which describes the history and milestones of the Progress Software Corporation [11] in detail. Besides that article, knowledge obtained by colleagues of mine have been used which have been developing software for the Progress OpenEdge platform for many years.

It all started in the 1970's when a small group of developers started the *Data Language Corporation* (DLC). The main product idea was a database platform and programming language combined into one, something which did not exist yet back in the day. This product was initially called *Relational Data Language* which got renamed to *Progress 4GL* later.

Programming languages can be classified using a generation specification system [4]. A first generation programming language (1GL) is a machine-level programming language, which means that it is a programming language at the machine code level were instructions are written by the programmer using binary encoding. A second generation programming language (2GL) is a programming language at the assembly level, which means that the programmer writes assembly code for an assembler instead of writing the machine language directly. This made writing programs for a system architecture already more convenient for the programmer. However, an assembly language still makes it difficult to write more complicated programs, because there is no abstraction at all. As Brooksheare et al. describe it: "The situation is analogous to designing a house in terms of

TScIT 37, July 8, 2022, Enschede, The Netherlands

boards, nails, bricks, and so on. [...] The design process is easier if we think in terms of larger units such as rooms, windows, doors, and so on." Hence, this is why third generation programming language (3GL) were created, which abstracts certain low-level features for the programmer such that it is easier to write more complicated programs by translating the written software to assembler instructions. Eventually, there was a demand for more domain-specific programming languages with even more abstraction, hence fourth generation programming languages (4GL) got introduced, which is also the reason why DLC named their language *Progress 4GL*, because their language is a database query language and hence can be categorized as a 4GL.

Because some other companies saw big potential in the language, their market share had increased rapidly over time. Hence, the name of DLC's main product *Progress* got well-known within the commercial software development community, which eventually lead to a rename of the company to *Progress Software Corporation* (PSC). Later, the programming language *Progress 4GL* got re-branded to *Progress OpenEdge Advanced Business Language*.

During the evolution phase of the platform, many features got introduced to the language. Features like SQL integration and support for client-server based architectures got introduced. Eventually it was also possible to write entire graphical user interfaces natively within their language. Only after many years, in 1995, object-oriented programming syntax got introduced because of the rise of popularity of other object-oriented programming languages like C++ and Java. Because object-oriented support got introduced later, many of the earlier introduced features have not been encapsulated with this object-oriented approach. This seems to be the main reason why the code completion of these older language features is not as integrated within the IDE provided by Progress as many of the other languages and their IDEs nowadays.

In the early 2000's, a former consultant of the Progress Software Corporation and her husband, Judy and John Green, started developing a tool called *Proparse* which provides analysis of Progress OpenEdge ABL code. Initially, *Proparse* was created as a proprietary tool, but after many years of its creation, it has been published as an open-source repository on GitHub. One of the applications of *Proparse* is another open-source tool called *Prolint* which is "a tool for automated source code review", and "reads one or more source-files and examines it for bad programming practices or violations against coding standards" [7].

2 PROBLEM STATEMENT

Because of the inefficient code completion, incorrect syntax highlighting and the absence of integration of documentation within the PDS, the integrated development environment for the Progress OpenEdge platform can be improved. In this research paper, the importance of autocompletion, syntax highlighting and code documentation are discussed and the issues given in section 1 have been solved by implementing an IntelliJ IDEA [21] plugin.

2.1 Goal

The main goal of this research project is to improve the overall developer experience of an Progress OpenEdge developer by creating a plugin which adds language support for Progress OpenEdge ABL for the IntelliJ IDEA. The plugin should provide code completion suggestions based upon context, provide correct syntax highlighting and integrate class and method documentation support.

2.2 Research questions

In order to reach the goal mentioned in section 2.1, the following main research question will be answered:

- How can the development experience of a Progress OpenEdge ABL developer be improved?

In order to answer this main research question, the following sub-questions will be asked:

(1) How can the code completion be improved compared to the Progress Developer Studio for OpenEdge?
(2) How can the syntax highlighting improved compared to the Progress Developer Studio for OpenEdge?
(3) How can the method documentation be more integrated into the IDE compared to the Progress Developer Studio for OpenEdge?

3 RELATED WORK

3.1 Related tools

A plugin for Visual Studio Code which implements ABL language support for Visual Studio Code has already been created [6]. However, this plugin does not provide the same level of autocompletion as the original PSD provides and also makes the same syntax highlighting mistakes as described in the introduction. The source code is available publicly which might be useful.

There is already an ANTLR2 language description of Progress OpenEdge ABL available [10], which originates from an old open-source ABL parser called Proparse [14] (this tool was also mentioned in section 1.1 about platform history). This Proparse tool has been reused and updated to ANTLR4 for a plugin called *Code Analyzer for ABL in SonarQube* (CABL) [33]. As the name already suggests, it performs code analysis on software written in ABL.

3.2 Code completion

A case study by Amann from 2016 shows how the Visual Studio IDE [24] is used in practice by 84 professional C# software developers [1]. The participants in this case study installed a plugin in Visual Studio which recorded all actions performed within the IDE. The case study showed that the interaction "code completion" was the most frequently performed interaction. This shows the importance of code completion in an IDE for software developers nowadays.

A study by Murphy in 2006 did a similar case study for Eclipse [25]. It also shows that within Eclipse the content assist is a frequently used interaction by Java developers, but less as described in the case study from Amann.

Hyvönen and Mäkelä consider the usage of semantic autocompletion instead of syntactic autocompletion in a broader perspective [17]. They mention the existing syntactic autocompletion applications, like Microsoft's *IntelliSense* in the *Visual Studio IDE* and word predictions on mobile phones. The authors raise the perspective by one level by looking at what improvements are possible for

autocompletion at the semantic level. This means that the technology should not just look at which characters have already been entered and give a good suggestion by applying this information to a given dictionary, but that it should also look at relations of any preceding and/or succeeding tokens in order to give a good autocompletion suggestion. Hyvönen et al. mostly look at existing applications of semantic autocompletion in the context of search queries and they discuss the different types of relations that can be used for this semantic autocompletion.

Kang et al. propose a new framework to be used for efficient query autocompletion, also in the context of search queries [23]. They mention that the traditional method of using a lookup-based approach by reusing queries and search terms a model has seen in the past performs well on common queries, but do not function well on unknown or rarely used search queries. In order to fix this issue, the authors have created a framework called *QueryBlazer* which uses a generative approach for autocompletion instead of the look-up approach mentioned earlier.

Muhammed et al. discuss a technique called *Context Sensitive Code Completion*, which as the name already suggests, uses the overall context in order to perform better suggestions [2]. This overall context includes any user defined identifiers like variable or method names, but also language keywords. The paper describes the structured process which is applied in order to determine the most relevant code completion suggestion to the developer by using this overall context.

A research article by Bruch et al. describes a few concepts which will likely be adopted in the future in the development field of integrated development environments (IDEs) [5]. It illustrates the concept of "IDE 2.0", where more intelligence is shared amongst software developers. The article mentions that currently, most code completion systems are using relative simple systems in order to determine the best code completion suggestion. There are systems which prioritize the auto complete functionality based upon context and how commonly certain application programming interface (API) calls are used. However, this can only be applied for certain specific APIs and require a lot of work to implement for every API. Hence, the paper describes that in the future, collective information might be used to improve the code completion within IDEs. This information can be collected by the code completion system in the IDE, and shared amongst other software developers anonymously in order to provide better code completion suggestions.

To conclude the information mentioned above, it can be stated that software developers use code completion features provided by IDEs very frequently. Also, the potential of semantic code completion suggestions has been discussed.

3.3 Syntax highlighting

In 2016, Beelders and du Plessis have performed a study where they compared the effectiveness of syntax highlighting by looking at the movement of the eye while reading and the time it takes for the eyes to fixate [3]. In this study, there was not a significant difference between highlighted and unhighlighted code. However, the students participating in this research did seem to prefer highlighted code over unhighlighted code.

A study performed by Sarkar in 2015 also used eye trackers to determine the impact of syntax colouring on the comprehension of code [34]. It showed that syntax highlighting did significantly reduce task completion time, but the effect does decrease depending on the experience of the programmer. A study by Häregård [16] also did not show a significant difference between highlighted and unhighlighted code and also suggested a similar correlation between the impact of syntax highlighting and programming experience.

A paper by Patrignani demonstrates the potential benefits which syntax highlighting might have when applied to research papers [28]. The paper describes in which cases syntax highlighting within research papers can be useful. It also describes some downsides to syntax highlighting: relying on syntax highlighting too much disadvantages colorblind people, and that papers are often printed in a black-and-white format. It argues that there are already tools available which can help the author of a paper with such syntax highlighting, but that it does not get used quite often because of the struggle of researchers adapting new technologies. Finally, the paper shows some examples on how to adapt such colours and syntax highlighting within LaTeX.

Using the above mentioned papers, it can concluded that syntax highlighting does not necessarily improve effectiveness, however most programmers do prefer highlighted code over unhighlighted code. This preference justifies the need of proper syntax highlighting for the plugin.

3.4 Documentation integration

Forward and Lethbridge have performed a survey in 2002 about the relevance of software documentation and tools [12]. In this survey, they asked 41 professional software developers about the way they use documentation and what they think is important about writing good documentation. One of the questions asked the to participants was to rate how important particular documents contribute to the overall effectiveness of code documentation in their opinion. Availability received on average a 4.35 out of 5, hence it can said that the participants seem to think that the availability of the documentation is important to the effectiveness of documentation.

As already mentioned in section 3.2, a research article by Bruch et al. shows the concept of "IDE 2.0" [5]. Besides future developments of code completion, the article also describes the possible future developments of code documentation, where documentation of commonly used APIs can be improved using collective intelligence. Users might for example be able to give feedback on existing documentation or provide new documentation for undocumented pieces. This concept already exists for bigger projects, where documentation is maintained by the software development community. However, the article mentions that there generally is a lack of participation by software developers, which might be solved once a developer is able to provide such feedback and suggestions directly from their IDE.

4 METHODOLOGY

In order to answer the research questions and complete the goal of this research project, an IntelliJ plugin which adds language support for the Progress OpenEdge ABL progamming language will

be implemented. In order to do this, the following tasks have to be completed:

- Writing a lexer. The lexer should read a file and group all characters into the respective tokens.
- Writing a parser. The parser should check if the tokens match the given grammar of the Progress OpenEdge ABL programming language.
- Adding syntax highlighting based upon the parsed tokens.
- Implement code completion functionality based upon the parsed tokens.
- Implement the ability to add type hinting for *handle* typed variables, such that more specific code completion can be provided for this handle.
- Implement documentation integration by parsing ABLDoc comments which are placed above methods and classes.

4.1 Quality assurance

In order to assure the quality of the plugin and this research project, the compatibility with certain Progress OpenEdge ABL source code files have been tested. A publicly available repository targeted for the Progress OpenEdge 12.2 version has been used.

Because there are not many open-source repositories available which can be used for testing, the *corelib* library from the Progress ADE Sourcecode repository has been chosen [9]. This code library is distributed with every Progress installation containing system defined data-structures and tools defined like collections, maps, JSON parsers and many other useful tools. It is quite broad and uses many of the object oriented language constructs.

5 ABOUT THE LANGUAGE

5.1 Procedural vs. Object Oriented

Initially, the Progress OpenEdge ABL language was created as a procedural data querying language. This means that all software is structured into procedures, and did not allow any abstraction like an object-oriented programming language would be able to do. Procedures are be defined in a *.p* and can have multiple input and output parameters. As mentioned in section 1.1 about the history of the Progress OpenEdge platform, the language eventually evolved into an object-oriented programming language by introducing *.cls* files, which can contain a class, interface or enumerator definition.

5.2 Keywords

The keywords in ABL work in a bit of a different way compared to most other programming languages: many keywords and built-in functions in the language can be abbreviated. If a keyword is abbreviateable, it also has a minimal length of characters which need to be present before the keyword is correctly recognized. For example, the keyword “define” has a minimal starting length of 3, so the keyword can be written as “def”, “defi”, “defin” or “define”. This makes writing a lexer for the language difficult, because an abbreviateable token can consist of many different character sequences. Also, some of the keywords are unreserved and hence can also be used as an identifier for a variable.

5.3 Handles

As already mentioned in section 1, the language has a data type called a “widget-handle”, or “handle” for short. A handle is a reference to one of the 73 available defined handle-based objects. They can reference to database tables (temp-tables, work-tables, datasources, datasets), system-level interactions (file reading, running procedures in other files, compilation) and graphical user interface (GUI) related tasks (rendering images, input boxes, windows, and many more).

6 IMPLEMENTATION

In this section, the implementation and structure of the plugin which has been written for the IntelliJ Platform SDK as a part of this research project will be discussed.[1]

6.1 Lexer and parser

As mentioned in section 3.1 about the existing alternatives, there was already an ANTLR language description available of the Progress OpenEdge ABL programming language. Because the language structure is quite big and complicated in terms of grammar rules, it felt unnecessary to redefine this grammar from scratch.

The IntelliJ Platform SDK examples which are given on the documentation website did not make use of an ANTLR-based grammar. Those examples made use of JFlex [22] in order to generate a lexer, and made use of JetBrains’ self-developed Grammar-Kit plugin [18]. So, initially there was an attempt in converting the existing Proparse [10] ANTLR-4 grammar to JFlex and Grammar-Kit. However, this was difficult, since ANTLR4 does allow for left recursive grammars, while the Grammar-Kit tool discourages the usage of left recursion in its grammars. This meant that many modifications in the existing grammar needed to be performed in order to remove all the left recursions.

As a result of this, the *ANTLR4 IntelliJ Adaptor* was given another try [27]. The adaptor is a tool which allows ANTLR4 grammars to be used within a language plugin for the IntelliJ Platform SDK. The Proparse repository did not make use of the lexing features provided by ANTLR, but used a custom lexer implementation defined in Java instead. Eventually, an own ANTLR4 lexer has been written by using the list of all possible keywords which were listed as an enumerator in the Proparse [10] project. This list of enumerators also contained the minimal starting length (explained in section 5.2) of every keyword. Using this list of enumerators, a script which would create the lexer definition of every keyword using the minimal starting length could be written. As an example of what this script did, the lexer definition of the keyword "define" can be found in listing 1. Although this approach is not desirable, there does not seem to be another way of defining abbreviateable tokens in an ANTLR4 grammar.

Listing 1. ANTLR4 lexer definition of the keyword “define”.

```
DEFINE : 'def' ('i' ('n' ('e')?)?)?;
```

Finally, lexing rules of elements like number expressions, string literals, identifiers, whitespace characters and comments needed

[1]The plugin and its source code are publicly available at the GitLab instance of the University of Twente: https://gitlab.utwente.nl/s2240122/openedgeplugin

to be defined. For the comments, additional lexing rules have been specified in the case of documentation comments, such that the information from these comments could be easily extracted and displayed for the implementation of the improved documentation integration.

6.2 PSI classes

The IntelliJ Platform SDK uses a program structure interface (PSI) [20] which parses the files in a convenient way, allowing easy traversal of the different parsed elements. For example, the PSI allows for easy traversal to neighbouring nodes, child nodes, parent nodes and can also recursively traverse the PSI tree looking for a PSI element with a certain element type.

In the *psi* package of the plugin, many custom defined PSI elements can be found. The *ANTLR4 IntelliJ Adaptor* by default parses every non-leaf node to a *ANLTRPsiNode*, with as element type the name of the parsing rule. Because the given *PsiTreeUtil* class can recursively find parent, sibling or child nodes of a given PSI element by class type, it was easier to define custom PSI elements. These custom PSI elements represent a specific part of the entire PSI tree and match to exactly one parsing rule from our grammar. Custom PSI elements are also needed in order to define certain methods specifically related to a certain language construct. For example, the custom PSI element *DefinePropertySubtree* represents the *defineProperty* grammar rule, which describes the definition of a property inside a class. Additional methods which have been implemented for this custom PSI element are for example *getName()* and *getType()*, which allow for easy access to the name and type of the defined property respectively. Also, methods like *getClassSubtree()* allowed for easy access to the *ClassSubtree* which this property belongs to.

6.3 Completion Contributor

The *OpenEdgeCompletionContributor* class takes care of the code completion within the editor. By calling the *extend()* method originating from the parent class *CompletionContributor*, completion suggestions can be provided for specific token types. In the case of our plugin, we have defined four of these completion contributor extensions:

- Method or attribute code completion after an identifier
- Method or attribute code completion after the *this-object* token
- Widget-handle type completion when defining a type-hint
- General code completion when invoking the code completion in any other context than defined above.

In order to reuse code completion suggestions, many custom PSI elements have a *getSuggestions()* method, which gets all suggestions of a certain variable or property. Also, many PSI elements have a *getLookupElement()* method, which returns a *LookupElement* representation of that PSI element such that the display of these elements are consistent throughout the different code completion contexts.

6.4 Documentation Provider

The *OpenEdgeDocumentationProvider* provides the integrated documentation when this is requested by the user. All elements which are able to display documentation should implement the custom defined *DocumentedElement* interface. This interface contains only one method, namely *getDocumentation()* which returns a string containing the respective documentation if available. The usage of such an interface is useful, because this allows other PSI elements to also inherit this interface in the future when documentation can be provided for such elements.

6.5 Data types

The *OpenEdgeDataType* enum contains a list of all possible data types in the Progress OpenEdge ABL language. In case of the class or handle data type, a more specific type definition is possible. Hence, the class and handle data types both contain a variable to specify this subtype, namely *className* and *handleType* respectively.

All handle types are defined in the *OpenEdgeHandleType* enum. The class *OpenEdgeHandleTypeDetails* contains a statically defined list of attributes and methods for every handle type, extracted from the documentation provided by Progress [31]. These method and attribute definitions are then used in the constructor of every handle type definition in the *OpenEdgeHandleType* enumerator.

7 RESULTS

7.1 Code completion

The plugin is able to introspect a code file and find all accessible variables, properties and methods and give code completion suggestions for these definitions. It also shows the type of a property or return type of a method. In order to get the code completion of a class, the class does need to be imported by adding a *using* statement at the start of the file. In figure 1, we can see a screenshot of what the code completion suggestions of a custom defined class called *Car* look like.

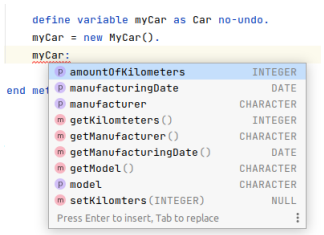

Fig. 1. An example of code completion for a class called *Car*.

In case a handle is defined and its "subtype" could not be deducted from its context, type hints can be used in order to manually define the subtype of a variable. It then shows all possible built-in functions and attributes of this subtype. In figure 2, the code completion suggestions for a *temp-table* handle can be seen.

Also, when invoking the code completion context when not trying to access a method or variable, it will show all properties, methods and local variables available within that context.

7.2 Syntax highlighting

As visible in figures 1 and 2, proper syntax highlighting has been implemented. In the case an unreserved keyword is used as an identifier, the syntax highlighter will update the syntax highlighting

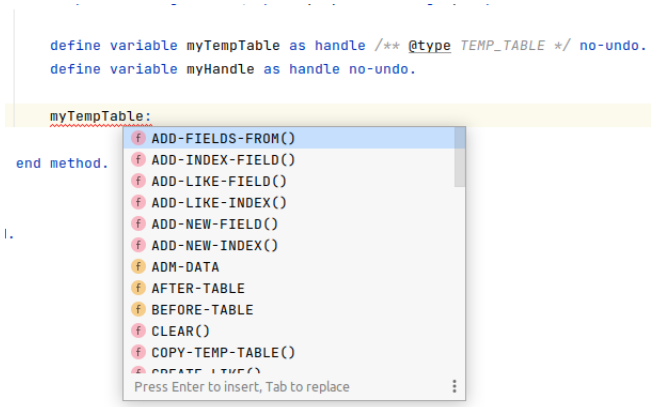

Fig. 2. An example of adding type hints to a handle

of the given keyword once it has detected using the context that it is indeed used as an identifier instead of an unreserved keyword.

7.3 Documentation integration

When performing the action *View | Quick Documentation* (by default using the shortcut *Ctrl+Q*) in the IntelliJ IDEA while selecting one of the code completion suggestions, the documentation of that method will show up if available. Using this feature, a software developer is easily able to see what a certain method does or what certain input parameters mean. In figure 3, the documentation of the *setKilometers()* method can be seen, which takes in one integer parameter as input. In the case no documentation is provided for a method, the IDE tells that there is no documentation available.

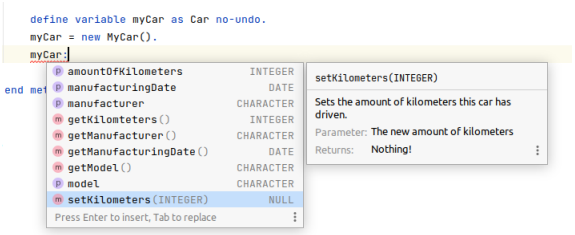

Fig. 3. An example of documentation integration

The nicely formatted documentation pop-up can also be shown when performing the same action in the editor itself at places where a method is used or defined.

7.4 Other features

During the development of the plugin, other useful features have also been implemented, like the following:

- Implemented a *Structure view*, which shows all methods and properties of a class
- Added a references for fields and import statements, allowing to easily navigate to the definition of the field or class.
- Error annotation when the parser was unable to parse the source file correctly.

7.5 Testing results

As described in 4.1, the Progress ADE Sourcecode [9] repository has been used in order to assure that the plugin works for most Progress OpenEdge ABL projects. All of the 230 source files were parsed perfectly fine, except for files where annotations were used. The Proparse [10] lexer definition used a pre-processor to filter out such annotations for the actual lexer, because annotations in Progress OpenEdge ABL are only used upon build-time, and are not available on runtime. Since this pre-processor for the plugin has not been implemented, it does not recognize the annotations and hence it results into being marked as an error in the IDE. To fix this, either the pre-processor could be implemented as an ANTLR4 lexer, or the annotations can be parsed as comments instead.

8 CONCLUSION

To conclude, an answer will be given to the main research question as proposed in section 2.2. In order to do this, the three sub-questions as proposed in 2.2 are answered first. Using the conclusions of these three sub-questions, a conclusion can be drawn for the main research question.

8.1 Research Question 1

How can the code completion be improved compared to the Progress Developer Studio for OpenEdge?

The code completion of the Progress Developer Studio for OpenEdge can be improved by adding type deduction and type-hints for handles to our plugin. This allows the subtypes of a handle to be defined and hence more selective code completion suggestions can be given.

8.2 Research Question 2

How can the syntax highlighting improved compared to the Progress Developer Studio for OpenEdge?

The syntax highlighting of the Progress Developer Studio for OpenEdge can be improved by ensuring that unreserved keywords which are used as an identifier are correctly identified by the IDE by using the semantics of such a token.

8.3 Research Question 3

How can the method documentation be more integrated into the IDE compared to the Progress Developer Studio for Open Edge?

The method documentation integration of the Progress Developer Studio for OpenEdge can be improved by recognizing code comments at the lexer level, so that this documentation can be shown in a nicely-formatted manner when the software developer might need it. For example, at method calls, variable definition or when using the code completer.

8.4 Main Research Question

How can the development experience of a Progress OpenEdge ABL developer be improved?

The development experience of a Progress OpenEdge developer can be improved by implementing code completion which is more selective for widget-handle objects, by implementing proper syntax highlighting and by integrating user-written method documentation

inside the IDE itself. However, the developer experience can be improved even more by implementing additional features to the plugin, as mentioned in section 9.

8.5 Threats to validity

As already mentioned in section 4.1, there are not many publicly available source code repositories of the Progress OpenEdge language. To improve the validity and stability of the plugin, the amount of repositories used should be increased such that the certainty that the lexer and parser are implemented correctly can also be increased.

The validity and stability of the plugin could also be improved by generating a set of test-cases for the grammar. Many examples already exist for this in the computer science literature, like generating test cases using the grammar [15] or in combination with "whitebox fuzzing" [13].

9 FUTURE WORK

In the near future, I am planning to add the following features to the plugin:

- Adding compiler integration allowing to compile source files.
- Using the compiler to annotate build-time errors inside the editor of the IDE.
- Adding type checks to the parser to detect type errors before attempting to compile.
- Improving the code completion even more by using semantics in order to give a better prioritization of the suggestions.
- Besides integrating the documentation of user-written functions, the documentation of system-level functions can be integrated too.
- Improving the validation by adding additional tests as mentioned in section 8.5.

REFERENCES

[1] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. 2016. A Study of Visual Studio Usage in Practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 124–134. https://doi.org/10.1109/SANER.2016.39

[2] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. 2014. CSCC: Simple, Efficient, Context Sensitive Code Completion. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 71–80. https://doi.org/10.1109/ICSME.2014.29

[3] Tanya R. Beelders and Jean-Pierre L. Du Plessis. 2015. Syntax highlighting as an influencing factor when reading and comprehending source code. *Journal of Eye Movement Research* 9, 1 (2015). https://doi.org/10.16910/jemr.9.1.1

[4] J. Glenn Brookshear and Dennis Brylow. 2020. *6.1 Historical Perspective*. Pearson, 320–326.

[5] Marcel Bruch, Eric Bodden, Martin Monperrus, and Mira Mezini. 2010. IDE 2.0: Collective Intelligence in Software Development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (Santa Fe, New Mexico, USA) *(FoSER '10)*. Association for Computing Machinery, New York, NY, USA, 53–58. https://doi.org/10.1145/1882362.1882374

[6] Camicas Christophe. 2020. *OpenEdge ABL*. Retrieved May 8, 2022 from https://marketplace.visualstudio.com/items?itemName=chriscamicas.openedge-abl

[7] Jurjen Dijkstra. 2001. What is Prolint. WebArchive. Retrieved May 20, 2022 from https://web.archive.org/web/20011109094234/http://www.global-shared.com/prolint/prolint.htm

[8] Eclipse Foundation n.d.. *Eclipse IDE*. Retrieved May 3, 2022 from https://eclipseide.org/release/

[9] Mike Fechner. 2016–2022. *ADE Sourcecode*. Retrieved June 22, 2022 from https://github.com/consultingwerk/ADE-Sourcecode

[10] Mike Fechner, Sebastian Wandel, and Marian Edu. 2014–2021. *Proparse*. Retrieved May 3, 2022 from https://github.com/consultingwerk/proparse

[11] Finland Progress User Group n.d.. *A History of Progress*. Retrieved May 23, 2022 from https://www.finpug.fi/historiaa/a-history-of-progress/

[12] Andrew Forward and Timothy C. Lethbridge. 2002. The Relevance of Software Documentation, Tools and Technologies: A Survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering* (McLean, Virginia, USA) *(DocEng '02)*. Association for Computing Machinery, New York, NY, USA, 26–33. https://doi.org/10.1145/585058.585065

[13] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

[14] John Green. 2007. *Proparse Book*. Retrieved May 3, 2022 from http://www.oehive.org/proparse/

[15] Hai-Feng Guo and Zongyan Qiu. 2013. Automatic Grammar-Based Test Generation. In *Testing Software and Systems*, Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–32.

[16] Erik Häregård and Alexander Kruger. 2019. Comparing syntax highlightings and their effects on code comprehension.

[17] Eero Hyvönen and Eetu Mäkelä. 2006. Semantic Autocompletion. In *The Semantic Web – ASWC 2006*, Riichiro Mizoguchi, Zhongzhi Shi, and Fausto Giunchiglia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 739–751.

[18] JetBrains. 2014–2021. *Grammar-Kit*. Retrieved June 14, 2022 from https://github.com/JetBrains/Grammar-Kit

[19] JetBrains 2022. *IntelliJ Platform SDK*. Retrieved May 8, 2022 from https://plugins.jetbrains.com/docs/intellij/welcome.html

[20] JetBrains 2022. *IntelliJ Platform SDK: What is the PSI?* Retrieved June 24, 2022 from https://plugins.jetbrains.com/docs/intellij/psi.html

[21] JetBrains n.d.. *IntelliJ IDEA: Capable and Ergonomic IDE for JVM*. Retrieved May 8, 2022 from https://www.jetbrains.com/idea/

[22] JFlex n.d.. *JFlex - The Fast Scanner Generator for Java*. Retrieved June 14, 2022 from https://www.jflex.de/

[23] Young Mo Kang, Wenhao Liu, and Yingbo Zhou. 2021. QueryBlazer: Efficient Query Autocompletion Framework. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining* (Virtual Event, Israel) *(WSDM '21)*. Association for Computing Machinery, New York, NY, USA, 1020–1028. https://doi.org/10.1145/3437963.3441725

[24] Microsoft n.d.. *Visual Studio: IDE and Code Editor for Software Developers and Teams*. Retrieved May 8, 2022 from https://visualstudio.microsoft.com/

[25] G.C. Murphy, M. Kersten, and L. Findlater. 2006. How are Java software developers using the Eclipse IDE? *IEEE Software* 23, 4 (2006), 76–83. https://doi.org/10.1109/MS.2006.105

[26] Oracle n.d.. *How to Write Doc Comments for the Javadoc Tool*. Retrieved May 8, 2022 from https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

[27] Terence Parr, Bastien Jansen, Danny van Bruggen, Yegor Petrov, and Alex Katlein. 2015–2021. *ANTLRv4 support in IntelliJ IDEs*. Retrieved June 14, 2022 from https://github.com/antlr/antlr4-intellij-adaptor

[28] Marco Patrignani. 2020. Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets. https://doi.org/10.48550/ARXIV.2001.11334

[29] Progress Software Corporation 2017. *Invoking syntax-completion assistance*. Retrieved May 3, 2022 from https://docs.progress.com/bundle/openedge-developer-studio-olh-117/page/Invoking-syntax-completion-assistance.html

[30] Progress Software Corporation 2017. *What is OpenEdge*. Retrieved May 3, 2022 from https://docs.progress.com/bundle/openedge-guide-for-new-developers-117/page/What-is-OpenEdge.html

[31] Progress Software Corporation 2021. *Handle Reference*. Retrieved May 6, 2022 from https://docs.progress.com/bundle/openedge-abl-reference-117/page/Handle-Reference.html

[32] Progress Software Corporation n.d.. *Introduction to Progress Developer Studio for OpenEdge*. Retrieved May 3, 2022 from https://www.progress.com/services/education/openedge/introduction-to-progress-developer-studio-for-openedge

[33] Gilles Querret. 2016–2022. *CABL - Code Analyzer for ABL*. Retrieved June 24, 2022 from https://github.com/Riverside-Software/sonar-openedge

[34] Advait Sarkar. 2015. The impact of syntax colouring on program comprehension.. In *PPIG*. 8.