Javardise: A Structured Code Editor for Programming Pedagogy in Java

André L. Santos
Instituto Universitário de Lisboa (ISCTE–IUL), ISTAR-IUL
Lisboa, PORTUGAL
andre.santos@iscte-iul.pt

ABSTRACT

The syntax of a programming language is the textual form – that conforms to a grammar – to express instructions of a programming model. The key idea of structured code editors is to constrain editing to syntactically valid program code, that is, the modifications ensure that the source code always conforms to the grammar. Syntax is considered an entry barrier when learning how to program. In this work we rehash the concept of structured code editors targeting programming education. We present Javardise, a structured editor for a subset of the Java language, and discuss its features in the light of programming pedagogy.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Applied computing** → *Interactive learning environments.*

KEYWORDS

Structured editors, programming pedagogy, Java

ACM Reference Format:
André L. Santos. 2020. Javardise: A Structured Code Editor for Programming Pedagogy in Java. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3397537.3397561

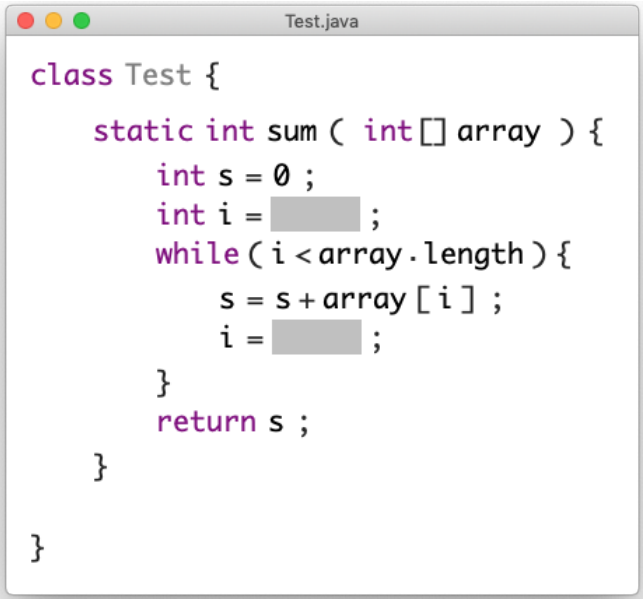

Figure 1: Javardise editor. Two placeholders (gray) yet to be filled with expressions.

1 INTRODUCTION

In Software Engineering there is a usual distinction between *essential* and *accidental* difficulties [2]. Essential difficulties are those that are inherent to the complexity of the problem at hand, whereas accidental difficulties emerge from the form that is used to map the problem into a solution. With respect to programming, language-independent algorithms are mapped to a particular programming language syntax. Understanding the algorithm and the general programming concepts are the essential difficulty, whereas the syntax is an accidental one. Mastering a programming language requires understanding its programming model, as well as the syntax to instantiate it. We may known well how a sorting algorithm works (essence), but not being immediately able to implement it using a particular language that we are not acquainted with (accident).

Although syntax is not the essence of a program, it may have a significant impact on the usability of a programming language. Particular characteristics of certain syntaxes may consist of hurdles, especially to novices [3, 11]. Regarding experienced programmers, although they should be able to understand every kind of syntax, they usually have strong feelings in favor or against particular kinds of syntax (as part of the so-called *programming language wars* [10]).

Syntax-directed code editing was proposed back in the 1980's [12], being the key idea the concept of a code editor that constrains the user editing activity so that program being written is (almost) always syntactically valid. That is, the source code, while it may have semantic errors, it always parses (i.e. it is accepted by the grammar). We refer to this kind of tool as a *structured editor.*

Structured editors stemmed from a line of research at Cornell and Carnegie Mellon Universities[1], and were adopted therein for teaching, as well as in other universities in USA during the 1980's [8]. Despite the promising idea, they have neither gained widespread adoption in programming education, nor in the software industry.

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03.
https://doi.org/10.1145/3397537.3397561

[1] From the available public information, structured editors are currently not being used in introductory programming courses at these institutions.

The reasons possibly relate to the usability of the editing experience, but this subject was not studied systematically.

Projectional editors, such as those that can be developed with MPS (JetBrains' Meta Programming System[2]), is a closely related tool concept that in addition to AST-based editing is also capable of combining multiple syntaxes. Nonetheless, the code writing experience of both structured editors and projectional ones is similar. Projectional editors have a few usability issues that may hinder their wide adoption [13]. However, a usability study [1] revealed that projectional editors are efficient to use for basic editing tasks – the case with introductory programming. The effectiveness of structured editors through controlled experiments has not been a subject of much research.

We believe in the potential of structured code editors, especially for introductory programming purposes. We aim at rehashing the concept, perform user studies to investigate how to maximize the usability of structured editors for introductory programming, and further evaluate their benefits and drawbacks. At this stage, we have developed Javardise[3], a structured editor for a subset of the Java language (see Figure 1), covering the syntax that we expose to students during the first programming course taught at our institution.

2 RELATED WORK

Several frameworks for generating structured editors have been proposed, namely GNOME [4] and MacGnome [8]. Whereas GNOME was based on menu interaction, not allowing direct text editing, MacGnome allowed temporary editing of plain text through a separate editing mode. The editors generated by these frameworks were based on a model-view-architecture, being the model the abstract syntax tree (AST) of the programming language. Our Javardise editor also follows this architectural pattern.

Barista [5] is a framework for developing structured editors, comprising innovations that enable additional visualizations to be embedded inside the editors. These might consist of complementary information (e.g., images) or alternative views for code (e.g., math notation). Barista is based on incremental parsing, freely typed text is converted into the model as the user edits the code.

Javardise consists of a custom-made editor for a particular syntax (Java), built without resort to a framework to generate editors. We are following this strategy to fully exploit the possibilities of a structured editor specifically tuned for a particular language, without compromising flexibility, as the genericity of a framework usually implies. From a wider programming pedagogy perspective, given that a small set of languages represents the vast majority of programming languages used in teaching [9], we argue that developing highly tuned editors for a small set of languages may be beneficial in contrast to relying on a generic framework.

The so-called block-based languages (e.g., Scratch [7]) are a modern approach to structured editors for education, and are currently being used extensively to teach computing concepts to early age apprentices. The block manipulations relief users from dealing with syntax errors. As opposed to general purpose languages, these languages usually have some domain-specificity. Block-based languages are considered an adequate entry point into computing. A study revealed that interaction modality (blocks vs. text) does have an impact on learners experience with programming, favoring blocks [14]. However, block-based languages have their drawbacks. Even for simple computations, they require a considerable amount of user interaction. Further, at some point learners will have to transition to actual source code.

Stride [6] is a Java-like language that was recently incorporated into the BlueJ[4] pedagogical programming environment. Stride is an editor where programs are edited by composing frames that hold instructions, as a hybrid form of editing between blocks and text. The frames enable innovative forms of presenting code. Some syntactical elements of Stride differ, albeit lightly, from Java. Given that it is a recent development, its effectiveness was not yet evaluated. Stride interaction is based on keyboard shortcuts or menus, not following the regular left-to-right flow of typing. In our approach we aim at an editor that embodies the actual syntax of Java, while providing, as close as possible, an editing experience with the look and feel of a conventional code editor. With Javardise users only type elements of the actual source in their regular ordering, albeit with constraints that disallow syntax errors and automate certain elements, driving the typing activity.

3 JAVARDISE

We developed a prototype of a structured editor that supports a subset of Java's syntax. Given that our aim is to aid programming pedagogy, and we do not support some non-elementary language features, such as type casts, imports, generics, and lambdas, as they are not essential for introductory programming.

3.1 Overview

The editing experience of Javardise is characterized by the user never seeing code that is syntactically invalid, except during transient moments while an identifier or literal is being written (visible in Figure 1). The appearance of the source code resembles what would be displayed in a regular code editor of an IDE when the code is automatically indented. Block brackets are automatically balanced and cannot be deleted, indentation is always right by design, and semi-colons automatically appear without having to be typed. The user may insert statements at valid locations inside a block by traversing the positions (lines) using the up/down keys or directly. Some tokens are editable (e.g., identifiers), whereas others not (e.g., modifiers). Some tokens are selectable (e.g., control structure keywords, for deletion), whereas others not (e.g., semi-colons, since their presence is merely syntactical).

We have designed the interaction so that the editing look and feel resembles as much as possible a text editor. The code can be fully edited just by using the keyboard, as opposed to block-based languages and Stride. Additionally, mouse can be used to position the cursor at an editing position or to activate a popup menu with context-sensitive editing suggestions.

The following sections detail how interaction is performed to insert elements in the source code. We use informal state machine-like diagrams where each state represents the generation of a part

[2] www.jetbrains.com/mps
[3] Java Rookies Driven Into a Structured Editor
[4] www.bluej.org

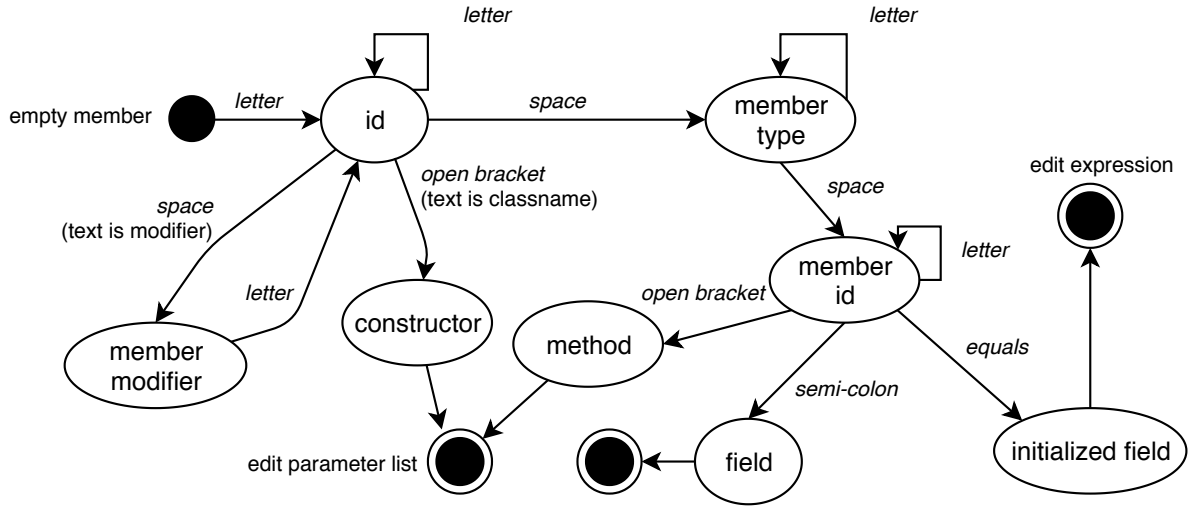

Figure 2: Process of inserting class members (fields, constructors, and methods) in Javardise.

of the whole element insertion, and each transition occurs when an allowed character is typed (annotation in the edge). Disallowed characters (not represented in any edge) are blocked from being typed. If the sequence of transitions does not reach a final state, then the whole insertion is cancelled, implying that the code is not modified. For the sake of brevity, we omit a few cases in the diagrams that may be easily inferred through analogous strategies to those presented.

3.2 Class Members

In Java, every file holds a class definition that contains several members, namely fields, constructors, and methods. Each of these members may have modifiers (e.g., `public`, `final`, etc). Figure 2 describes the process for inserting a class member. Starting from an empty line, the user may start typing letters to form an identifier. Once space is hit when the current text is a modifier, such modifier is added to a list, and a new identifier starts to be defined. This list of modifiers grows until another path is taken, and will be used when inserting the member declaration.

If an opening bracket is typed and the current text is the same as the class name, a constructor is inserted and the user proceeds to edit its parameter list. If space is hit when the current text is not a keyword, the member type starts to be defined with a sequence of letters. Once the member type is defined, its identifier is analogously defined. At this point, insertion will result either in a method or a field. If an opening bracket is typed, a method is inserted and the user proceeds to edit its parameter list. If a semi-colon is typed, a field is inserted without initialization. Finally, if equals is typed, a field is inserted and the user proceeds to edit the expression.

3.3 Expressions

Expressions are used in variable assignments, control structures, arguments, and return statements. The simplest expressions are literals (e.g., numbers, boolean values, or strings) or variable expressions. When typing an expression, if the first character is a digit or a dot, then a number literal is inserted upon typing the remaining digits. String literals are inserted when a quote is first typed, followed by a sequence of arbitrary characters and a closing quote.

If the user starts to type letters, the path goes towards inserting a boolean literal, variable expression, member expression, or static method call. A sequence of letters followed by enter, results in a boolean literal if the current text matches "true" or "false", or in a variable expression otherwise. If a dot is typed along the process, the expression becomes a member expression. In this case, upon typing the member identifier, it is inserted a field expression if a dot is typed (possibly chaining other fields), or a method call if an opening bracket is typed. Finally, a static method call is inserted if an opening bracket is typed while defining the first identifier. These last two cases end by having the user inserting a list of arguments (expressions).

Binary expressions, for arithmetic, logical and relational operations, are obtained when typing a valid binary operation having the cursor is placed at the right side of another expression. This inserts a binary expression with the current expression as the left part. The user proceeds to edit the right part of the binary expression.

We allow bracketing on a binary expression, implying that brackets are placed around the left and right parts of the nearest binary operator. Bracketing is a common challenge with projectional editors [13], and is still limited and needing further work to be seamlessly used.

Unary expressions, such as logical or numeric negation, are obtained by typing a valid unary operator when the cursor is at the left of another expression. This causes the unary operator to be applied to such expression. The cursor position being at the right or left is an important detail, given that some symbols are valid as both binary and unary operators (e.g., minus sign).

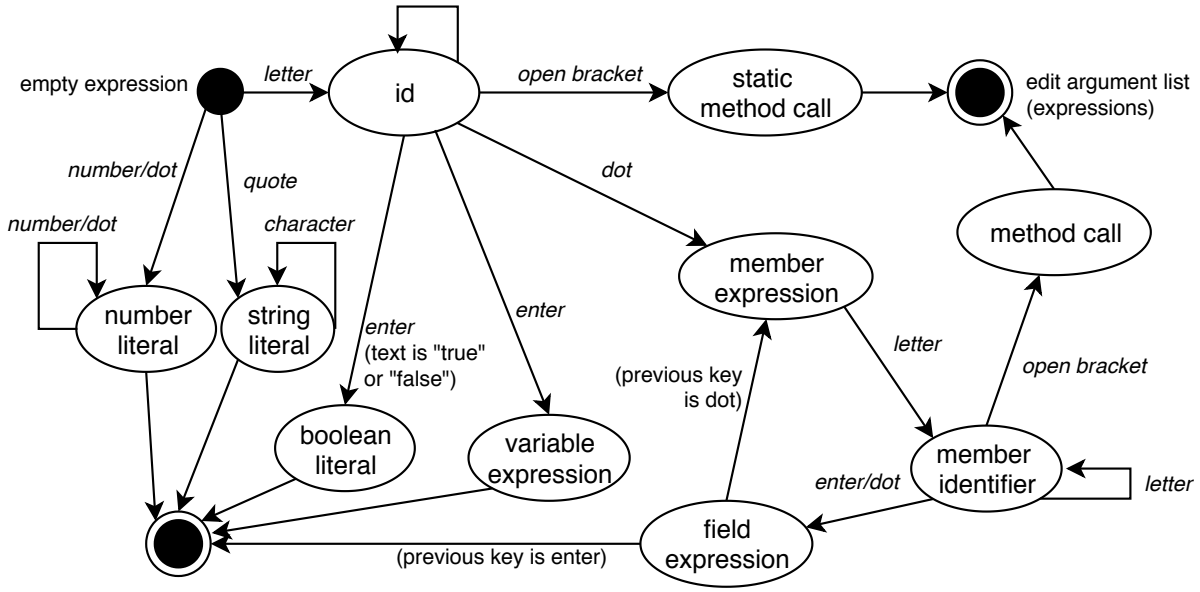

Figure 3: Process of inserting expressions in Javardise.

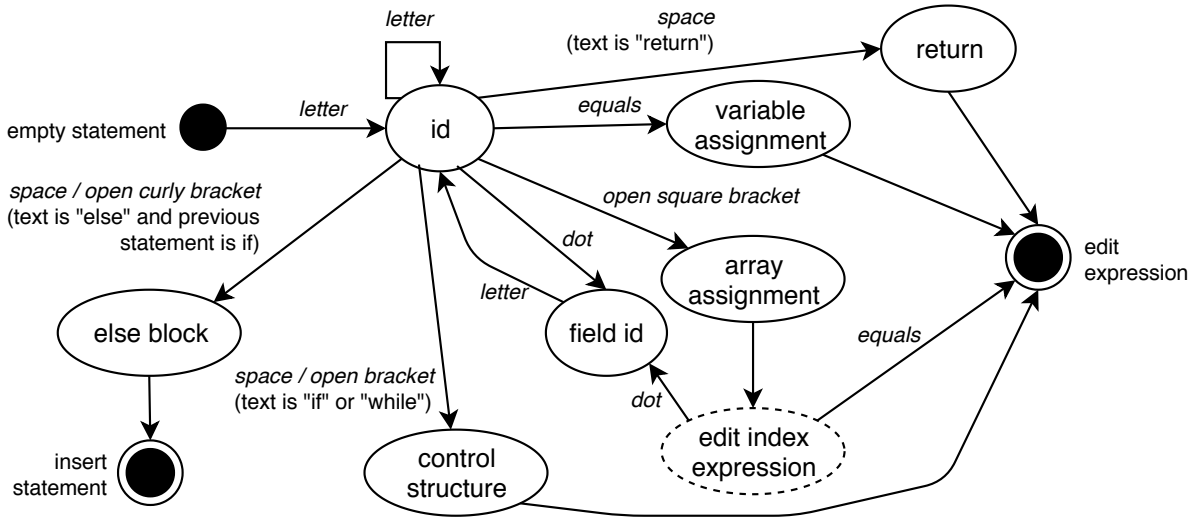

Figure 4: Process of inserting statements (assignments, return) and control structures (if-else, while) in Javardise.

3.4 Statements and Control Structures

The body of methods and control structures consists of a sequence of statements and control structures. Elementary statements are variable assignments (including array positions) and return, whereas the `while` loop and `if` (with the optional `else` block) are the elementary control structures. The user may insert these elements inside a block on an empty line, which is created by pressing enter on any line. According to what is typed by the user, different statements or control structures are inserted (see Figure 4). The figure omits the case of variable declarations with assignment, a case is similar to assignments.

Typing begins with an identifier, which may lead to a control structure or a statement. A control structure is inserted if the text matches "if" or "while" followed by a space or an opening bracket, and editing proceeds to insert the expression of the guard. The `for` loop, albeit slightly more complex, is handled in a similar way than a `while`, differing that the user proceeds to edit the variable initialization section. If the text matches "return" followed by a space, a return statement is inserted and editing proceeds to insert

its expression (which may remain empty). If the text matches"else", the previous statement is an `if`, and the user hits space or an opening curly bracket, an `else` block is inserted and editing proceeds to insert statements therein.

In case none of the previous cases applies, the inserted identifier is treated as a variable to be assigned. If a dot is typed, then the process of forming a field assignment starts. In case an opening square bracket is typed, an array position is being accessed and the user is led to edit its index expression. The assignment process ends when equals is typed, and editing proceeds to insert the expression.

3.5 Comments and Empty Space

In a language like Java, empty space (spaces, tabs, empty lines) has no effect on program semantics, and hence, it is ignored when parsing. Comments are also discarded when compiling. Empty space and comments are merely presentation aspects, but nevertheless, they are important for readability of the code.

As explained, our editor always maintains the right indentation and allows users to insert empty lines. These lines may remain empty, or a comment may be added using its syntax ("`//`" followed by arbitrary characters in the same line). Anything else other than a comment is discarded, unless it is captured in the insertion processes explained previously.

3.6 Deleting and Copying

Given that our editor maintains the source code syntactically valid, delete operations have to behave accordingly. Elements are deleted as a whole, namely a field, statement, parameter, control structure (and all the statements therein), or method (and its body).

We considered a few convenient operations for converting elements. A `while` may be switched to an `if` (and vice-versa), without having to delete the whole structure. Further, a variable declaration with assignment may be turned into a simple assignment (and vice-versa).

Deleting tokens that result in syntactically invalid code is not possible, as for instance, semi-colons and block brackets. The user may delete in isolation modifiers and parts of an expression. If an expression is left empty, its placeholder is left with gray background (as in Figure 1) for the user to fill-in. This actually consists of invalid syntax, and we discusse this issue in the next section.

3.7 Serialization

Serialization is performed as text to the actual source code visible in the editor. The only situation when the editor is displaying syntactically invalid code is when having empty placeholders. In this cases, we use a reserved identifier `$empty$` to denote an empty expression, while always outputting syntactically valid code. Deserialization is performed by parsing the code and populating the model, and the `$empty$` identifiers will be presented as in Figure 1.

We only support the syntax covered by the editor, so that all the source that was serialized with Javardise can be edited again using it. If a particular formatting (indentation, etc) is present in the source (possibly "manually" typed), it will be lost when edited and the file is saved.

The format used in projectional editors is typically not the actual source code. This is a well-known drawback of these editors, because the integration with source code control systems and related text-based tools is hard or limited. We decided to serialize the editor content to source code in a WYSIWYG fashion. In this way, that source can be used in versioning systems and further edited in other editors seamlessly (as students move forward).

3.8 Syntax Levels

Syntactical constructs can be grouped into categories (sequence of "levels"). In this way, the editor may be configured to support a minimal set of syntactical constructs at early stages, and progressively being used with increased levels that unveil new syntax possibilities as learning progresses. For the teaching progressing of the CS1 course at our institution we would use the following levels:

(1) Primitive types, return statements and expressions;
(2) `if-else` and method calls (recursive computation);
(3) Assignments and `while` loop;
(4) Arrays and `for` loop;
(5) Objects and reference types;
(6) Class definition.

Given that teaching strategies and curricula differ across institutions and courses, the grouping of syntactical constructs could be formed in different ways.

3.9 Implementation Notes

In informal experiments using early prototypes of Javardise, we realized that fine-grained interaction aspects, such as text caret positioning, widget traversals, as well as keystrokes to control these, were of major importance to editor usability. This was also noticed in a user study using MPS [1].

We decided to implement a custom-made editor, that is, without relying on any editor generation frameworks. In this way we are in full control of interaction possibilities, and we were allowed to implement features such as the syntax levels described previously. Although we are following a language-dependent approach, we argue that the interaction processes can easily be ported to other languages with the same kind of syntax (e.g., C).

Javardise was implemented in Java, using the SWT library[5]. This library is cross-platform, and the widgets are rendered by means of native primitives of the target operating system. Every visible element in the editor is a widget, either a text item or a composite widget that holds other items. We used a model-view-controller architecture, where an in-memory model represents the abstract structure of the program, which the view renders through the widgets.

4 DISCUSSION

The main goal of our approach is to avoid the syntax barrier at early stages of introductory programming, so that novices are not obstructed by accidents, while focusing their attention on program semantics. Javardise covers the syntax requirements of the first programming course taught at our institution. Despite informal experiments where a few users were asked to try the editor, no

[5] www.eclipse.org/swt

controlled experiments were carried out, neither the editor has yet been in used in classrooms. This section discusses the hypothetical benefits we aim at along with a forecast of drawbacks.

Malformed Code. Those who teach lab classes know that a considerable number of times when they are asked for help is due to a purely syntactical error/typo. For instance, a missing semicolon (or an accidental one placed right after a control structure guard), unbalanced brackets, using illegal characters in identifiers, "reversed" assignment statement, and numerous creative forms of syntax guessing. Even for an expert, finding the cause of the problem might not be immediate, as they are obfuscated in the code (often not properly indented). These issues contribute to slow-down the pace of a lab class. On the other hand, while in a lab class a student may have the aid of a teaching assistant or a colleague, syntactical hurdles may constitute a harder barrier to progress when alone. As syntax is not part of the essential difficulty of learning how to program, we argue that the associated hurdles should be minimized in favor of concentrating the effort on the essence.

Focus on Semantics. Part of the difficulty on learning programming, often described as "overwhelming", might be due to having to learn *simultaneously* a programming model and how to instantiate it using a particular syntax. We believe that once a solid understanding of semantics is gained, learning a syntax becomes a minor issue. The less time spent on syntax hurdles, the more time is spared to dedicate to semantics.

Switching to Conventional Editor. The fact that when using a structured editor the user is not typing every character of the source code may require an adaptation phase when switching to a conventional editor. This issue should be investigated. In an editor like Javardise, the code is being presented exactly as the actual source code, and hence, we hypothesize that users will likely get accustomed to the syntax. Nevertheless, it is worth noting that modern IDEs also perform various code insertions automatically and not every character is typed. Therefore, we do not expect major issues with respect to editor switching.

Stepwise Acquaintance with Syntax. We argue that exposing novices to more syntax than necessary to teach fundamental concepts is counter-productive. Syntactic variations ("sugar") at early stages may confuse the learner, since different variations of expressing the same thing (semantics) are presented. For instance, in a language like Java, there are five basic ways to express a variable incrementation. We believe that syntactical variations, such the possibility of using a `for` loop in place of a `while`, should be presented only when elementary constructs are well-understood (that is, intensely practiced). By this time, the learner is likely to better realize why such syntactic possibilities were invented, while not confusing the essence. An editor like Javardise, which supports syntax levels, is useful to this teaching strategy, as syntactical constructs may gradually be activated as learning progresses.

Unsatisfactory Usability. We believe that the biggest threat to Javardise, as well as to structured editors in general, is the lack of usability that may be caused by the rigid mode of editing the source code. Without a pleasant and productive means to manipulate the source code, the previous hypothetical benefits and the overall goal of easing introductory programming become hindered. Hence, effective structured editors should require a serious investment on the usability of the interaction. We plan to conduct qualitative studies to improve the usability of Javardise, as well as controlled experiments to evaluate its effectiveness.

ACKNOWLEDGEMENTS

I would like to thank the anonymous reviewers for their valuable comments on earlier drafts of this paper. This work was partially funded by *Fundação para a Ciência e Tecnologia* (FCT / Portugal) under the sabbatical research grant SFRH/BSAB/150483/2019.

REFERENCES

[1] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 763–774. DOI: http://dx.doi.org/10.1145/2950290.2950315

[2] Frederick P. Brooks. 1995. *The Mythical Man-Month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the Syntax Barrier for Novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 208–212. DOI: http://dx.doi.org/10.1145/1999747.1999807

[4] David B. Garlan and Philip L. Miller. 1984. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. *SIGPLAN Not.* 19, 5 (April 1984), 65–72. DOI: http://dx.doi.org/10.1145/390011.808250

[5] Andrew J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*. Association for Computing Machinery, New York, NY, USA, 387–396. DOI: http://dx.doi.org/10.1145/1124772.1124831

[6] Michael Kölling, Neil C. C. Brown, Hamza Hamza, and Davin McCall. 2019. Stride in BlueJ – Computing for All in an Educational IDE. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 63–69. DOI: http://dx.doi.org/10.1145/3287324.3287462

[7] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article Article 16 (Nov. 2010), 15 pages. DOI: http://dx.doi.org/10.1145/1868358.1868363

[8] Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. 1994. Evolution of Novice Programming Environments: The Structure Editors. In *of Carnegie Mellon University*. 140–158.

[9] Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. 2018. Language Choice in Introductory Programming Courses at Australasian and UK Universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 852–857. DOI: http://dx.doi.org/10.1145/3159450.3159547

[10] Andreas Stefik and Stefan Hanenberg. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 283–299. DOI: http://dx.doi.org/10.1145/2661136.2661156

[11] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Trans. Comput. Educ.* 13, 4, Article Article 19 (Nov. 2013), 40 pages. DOI: http://dx.doi.org/10.1145/2534973

[12] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (Sept. 1981), 563–573. DOI: http://dx.doi.org/10.1145/358746.358755

[13] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings (Lecture Notes in Computer Science)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.), Vol. 8706. Springer, 41–61. DOI: http://dx.doi.org/10.1007/978-3-319-11245-9_3

[14] David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Trans. Comput. Educ.* 18, 1, Article Article 3 (Oct. 2017), 25 pages. DOI: http://dx.doi.org/10.1145/3089799