Jasay: Towards Voice Commands in Projectional Editors

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

Alexandre Cancelinha
alexctra@hotmail.com
Instituto Universitário de Lisboa
(ISCTE-IUL)
Lisboa, Portugal

Fernando Batista
fernando.batista@iscte-iul.pt
Instituto Universitário de Lisboa
(ISCTE-IUL), INESC-ID
Lisboa, Portugal

ABSTRACT

Permanent disabilities or temporary injuries (e.g., RSI) hinder the activity of writing code. The interaction modality of voice is a viable substitute or complement for typing on a keyboard. This paper describes the design of Jasay, a prototype tool that enables developers to write Java code using voice commands. Our implementation relies on a third-party speech-recognition system to convert the voice into text. In turn, such a text is translated into commands that transform the abstract syntax tree (AST) of the code being edited. Jasay works as an extension to a projectional editor, taking advantage of having the abstract syntax tree always available without parsing, a permanent well-formed structure of the code, and unambiguous editing locations (e.g., class member, statement, expression, etc). An early experiment with Jasay involving 5 programmers has shown encouraging results, as they were able to perform small program modifications within reasonable time.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**; • **Software and its engineering** → **Development frameworks and environments**.

KEYWORDS

Programming, voice, projectional editors, Java

ACM Reference Format:
André L. Santos, Alexandre Cancelinha, and Fernando Batista. 2024. Jasay: Towards Voice Commands in Projectional Editors. In *2024 First IDE Workshop (IDE '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643796.3648449

1 INTRODUCTION

Using the computer for more than 5 hours a day is enough to accumulate stress in certain parts of the body such as shoulders, elbows, wrists, hands, and fingers due to the repetition of movements. These diseases are known as repetitive strain injuries (RSI), and the recovery time of this type of injury varies, with some taking from 3 to 6 months to recover fully [13]. Another study [1] showed that 41% of workers in a telecommunications company, who were in front of a computer for more than 5 hours a day, felt fatigue in the upper back area around the neck and 38% in the shoulder area. Voice is an alternative modality to type code that may alleviate programmers with RSI issues. Possibly not as a full replacement, but rather to relieve users from some keyboard/mouse activity.

Coding using voice modality has been a subject of research (e.g., [5, 7, 9, 14, 19]). Previous works consist of extensions of traditional IDEs, while most approaches translate voice commands into Abstract Syntax Tree (AST) transformations. Naturally, this strategy is easier when compared to lower-level text-edit transformations. To our knowledge, there are no previous approaches that exploit the integration of coding by voice in projectional editors. This work aims at providing voice modality to input code, alleviating programmers of being fully dependent on their hands for typing code on the keyboard.

This paper describes the design of Jasay, a prototype that achieves this goal by extending Javardise [15][1] – a projectional prototype editor for Java that uses JavaParser's [18][2] ASTs as the model for representing code. Jasay translates text given by a speech recognition system into AST transformation commands, taking into account the context of the editing location. Furthermore, we consider both string similarity and homophony when matching identifiers in scope. For example, if a user dictates a command "hip not zero" (not well pronounced), the command may take effect when the cursor is at the guard of a control structure whose scope has a variable named "heap". Our design exploits the possibility of a projectional editor obtaining a precise location of the cursor that "knows" what sort of tokens to expect (e.g., class members, statements, expressions).

2 PROJECTIONAL EDITING

In contrast to traditional text-based editors, which directly manipulate characters in a text file, a *projectional editor* [17] works with an underlying model, such as an abstract syntax tree (AST) or another representation. The model is projected in views, as in the principles of a model-view-controller architecture. The user interaction with the views originates commands that modify the model, whose changes are in turn reflected in the views. What the user sees does not necessarily resemble the storage representation. For instance, when using Jetbrains MPS[3] to write Java [10], the user sees actual Java on the screen, but the storage is in the tool-specific format for model serialization. On the other hand, in an editor such as Javardise [15] the storage format consists of regular Java code.

Although projectional editors have never reached the mainstream, their architecture has a major advantage. The code being written is never in an unparseable or partially parseable state, because there are no parsing processes while the code is being written. The model is at all times a well-structured representation of the

This work licensed under Creative Commons Attribution International 4.0 License.

IDE '24, April 20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0580-9/24/04.
https://doi.org/10.1145/3643796.3648449

[1] github.com/andre-santos-pt/javardise
[2] javaparser.org
[3] www.jetbrains.com/mps

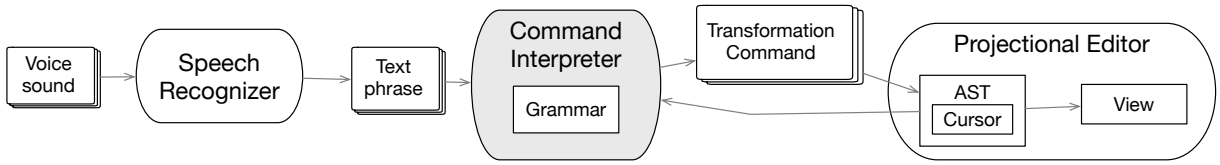

Figure 1: Architecture overview of Jasay. Arrows represent data flow.

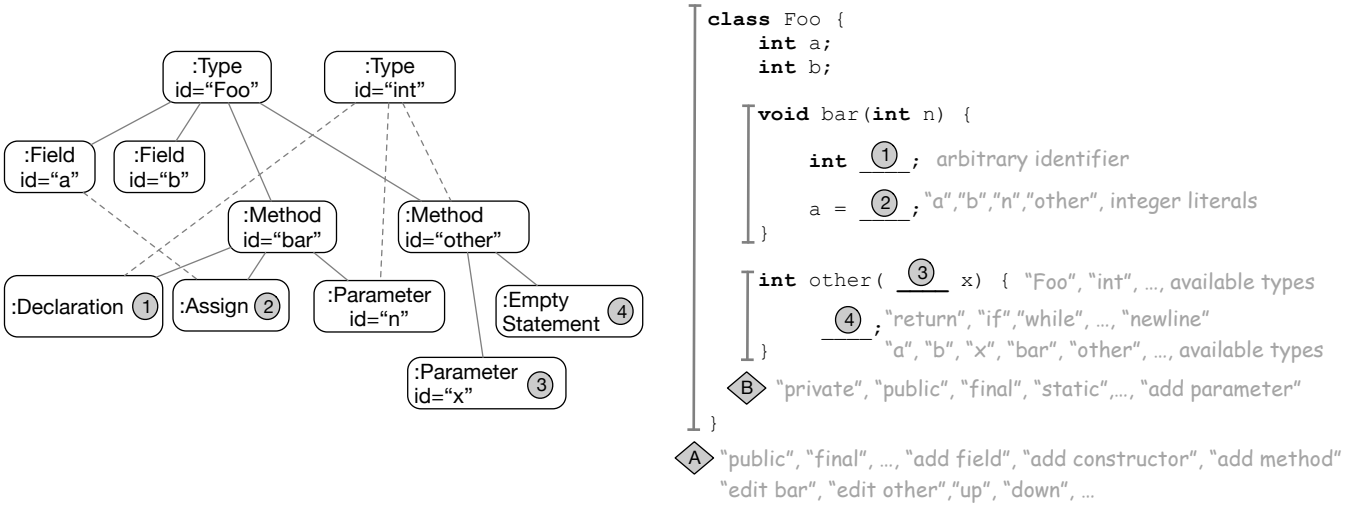

Figure 2: Editing contexts for voice commands in Jasay. In-situ: identifier (1), expression (2), type (3), statement (4); Ex-situ: class (A), method (B).

code. Although models might be in a semantically invalid state, they are always consistent with the views and can be fully analyzed at every moment. Every change to the model can be represented with a well-defined command, whose execution will never compromise the structure of the model. This facilitates programmatic manipulation of the code through commands that transform its model (e.g., AST).

When compared to text-insert commands, commands over an AST are at a higher abstraction level. For example, in text-based modifications, adding a parameter to an existing function *f*, could be expressed with <offset, string to insert> (e.g., <412, ", int param">. On the other hand, the equivalent AST command could be expressed with the <method, type, identifier>, where the first is a reference to the actual model element of the function, the second is a reference to an actual type, and the third is the identifier in isolation (as opposed to the text-based command).

In a projectional editor, the editing location (cursor) consists of a UI control that knows with precision what to expect in that location, for example, an arbitrary identifier (e.g., function name), an expression (e.g., control structure guard), or a statement (e.g., an empty line inside a block). This is an advantage we exploit in our approach because the editing context information can be used to constrain the set of valid commands to alter the code, leading to a reduced recognition space, which varies according to the editing location.

3 APPROACH

Figure 1 presents an overview of the main components of our approach. The core component developed in this work is the *Command Interpreter*, which stands in between a third-party Speech Recognizer that outputs text phrases from voice, and a Projectional Editor that exposes the ASTs of the files being edited. Furthermore, such an editor provides the location of the cursor as an AST node and allows for the execution of AST transformations.

Our speech interpretation only deals with ASTs, and when these are modified, the changes are reflected in the editor view due to the model-view-controller architecture of the projectional editor. Every command takes into account the editing context, which is provided as an AST node where the editing cursor is located. There are two categories of voice commands: *in-situ* or *ex-situ*. Figure 2 illustrates the different types of editing contexts using a code snippet and the respective AST. Blank spaces correspond to editing placeholders of the Projectional Editor, and the annotations in gray enumerate the tokens that could be inserted at the different locations. The numbers indicate contexts for in-situ commands, whereas the letters refer to the scopes for ex-situ commands.

In-situ commands consider cursor location in detail and their transformations actuate at the cursor location. They resemble what would be otherwise typed on the keyboard, despite omitting syntactic elements such as brackets and semi-colons. These commands are constrained by the cursor location in a fine-grained manner. For instance, inserting an integer is only possible if the cursor is

at an expression location, or inserting a return instruction is only possible if the cursor is inside a statement block.

Ex-situ commands may not correspond to what the user would be typing on the keyboard, and their transformations might not take place at the cursor location. The editing context of these commands is fairly broad, in the sense that the cursor location is not considered with as much detail as in in-situ commands. For instance, an "add field" command is available if the cursor is within a class body, or an "add parameter" command is available if the cursor is within a method. Both these cases span large sections of the code; hence, they are enabled almost at all times.

The *Command Interpreter* holds a grammar for recognizing valid commands, including those for navigating in the code, that do not alter the AST but instruct the editor to move the cursor. Despite having similarities with a traditional programming language grammar, its rules reflect code dictation in spoken language, as in previous approaches (e.g., [3, 5, 7, 19]). This implies that some syntactical elements, such as control structure brackets and semi-colons, are not part of the grammar rules. For example, a phrase like "if i equals 0" would match a valid if-statement [`if(i == 0)`]. Another important difference relates to how identifiers are handled, given that in programs they often consist of a concatenation of words that are spelled in isolation. For example, the phrase "set visible true" may translate to a method call [`setVisible(true)`].

Each grammar rule has an associated predicate to determine if it is applicable in the current editing context, provided by the cursor location. The grammar recognizes commands regardless of the latter having broken references due to misspelled or misrecognized identifiers. The command is analyzed to check for broken references, and we attempt to fix those *a posteriori*, by searching for approximate matches in the set of possible identifiers given the command context. We use two methods in combination to achieve this, string distance (Levenshtein) and phonetic similarity. When facing an unresolved identifier, we check for possible identifiers that slightly differ textually, and if there is none, we further convert the possible identifiers to graphemes and attempt to match them phonetically.

4 IMPLEMENTATION

We developed the Jasay prototype as a proof of concept of our approach. We extended Javardise [15] — a projectional editor supporting a subset of Java targeting educational purposes. Javardise runs on the JVM and it is implemented in Kotlin. Jasay was also implemented in Kotlin and is loosely coupled with Javardise, given that the former only gains access to the ASTs of the files being edited and to the cursor position. Due to the model-view-controller architecture of Javardise, any modifications to the ASTs are reflected in the views without requiring further procedures. The AST transformations are represented as commands [6].

The model representation in Javardise is based on the ASTs provided by the JavaParser [18], an open-source library written in Java that offers APIs for analyzing, manipulating, and generating Java source code. One of its main use cases is code analysis, where JavaParser analyzes source code and builds an AST that can be traversed to resolve reference types and collect information such as identifiers in scope. Javardise provides the cursor location as a JavaParser's AST node.

```
IfStatement: 'if' Expression?
Expression: Variable|Call|Integer|BiExpression|...
Variable: WORD+
Call: WORD+ Expression*
BiExpression: Expression? BIOPERATOR Expression?
Integer: 'zero'|'one'|'two'|...
WORD: [a-z]+
BIOPERATOR: 'plus' | 'minus' | 'equals' | ...
...
Add: 'add' ('field'|'method'|'parameter'|'setter'| ...)
Modifier: 'public'|'private'|'static'|...
```

Figure 3: Excerpt of grammar rules for handling in-situ and ex-situ voice commands.

Our *Command Interpreter* uses a context-free grammar built with ANTLR [8][4] for treating the text phrases. The grammar is not intended to be used to process a whole document, but instead, to attempt to match isolated sentences by matching one or more of its rules. The ANTLR nodes that result from rule matches are translated into conmmands that transform JavaParser's ASTs.

Figure 3 presents an excerpt of our rules for both in-situ and ex-situ commands. The grammar rules are naturally related to those of the programming language syntax. A study on how software developers verbalize code [3] (leading to the definition of Spoken Java) elicited a number of characteristics of spoken programs, to which our grammar adheres, namely:

- numbers and operators are in written form, to match the words of the text phrases;
- some syntactical elements (such as brackets and semicolons) are absent because we do not spell them when reading code;
- identifiers of variables or methods are formed by a sequence of one or more words ignoring capitalization, which will be concatenated to form CamelCase identifiers;
- some rules may allow missing terms (such as expressions), to support both atomic commands (e.g., "if") and compound ones (e.g., "if x equals 0").

Although capturing natural speech may be appealing from a user's standpoint, it implies difficulties regarding the ambiguity of rule matching, once more than one rule is matched. For example, "if isSelected" matches both rules IfStatement (Variable) [`if(isSelected)`] and IfStatement (Call) [`if(isSelected())`]. In these cases, we attempt to select among the alternatives by resolving name references to check which instruction would likely be the desired one. For example, if it exists a method `isSelected`, the second option would prevail over the first one.

Another issue relates to the multiple matches involving ex-situ commands because they may overlap in-situ ones. For example, "add parameter" matches both the ex-situ rule `Add` and the in-situ rule `Call`. In these situations, the in-situ command is attempted first, and if it results in broken references (i.e., there is no method called "addParameter"), the ex-situ command is executed instead.

When editing expressions, if the AST node of the cursor location is not empty, it is used to fill in terms of the grammar rules. For

[4] antlr.org

example, when spelling a voice command "plus one" on `i = x`, the `x` expression (cursor location) is used as the left expression of the `BiExpression` rule.

We have experimented Whisper [12] as the Speech Recognizer. This model is implemented as an encoder-decoder transformer, and was trained with 680,000 hours of data in several languages from supervised data that was collected from the web. This large amount of data increases the robustness of the system in terms of accent, background noise, and technical language, allowing also to perform transcripts in several languages. The Speech Recognizer works in a separate process running in Python, given the convenience of the available API to date. Since the *Command Interpreter* only relies on raw text phrases, the backing speech recognition technology could be easily exchanged. Phonetic similarity is checked using grapheme-to-phoneme transformations provided by the Gi2Pi Python library [11] (running in a separate process as well).

5 EARLY USABILITY EXPERIMENT

We carried out an early user experiment to assess if the approach is feasible and to collect usability issues. We recruited 5 programmers from our personal contacts, aged between 23 and 35, all of which with experience with Java. Participants were given 8 tasks that consisted of performing small increments to an existing Java file. Participants did not have any tool training session before the tasks.

Table 1 presents the experiment tasks and the average completion times in minutes. All tasks were successfully completed by every participant. However, the obtained times were clearly slower than if using a keyboard. We observed that a great part of the task time was spent on guessing commands, including those for navigation. We believe that after a brief training period, participants could become significantly faster in accomplishing this sort of tasks.

	Task description	Avg. time
1	Create a field named *age* of type *int*.	1:02
2	Create a private field named *id* of type *int*.	0:23
3	Create a public method named *factorial* that returns an integer.	0:33
4	In the *test* method, add an integer parameter named *number*.	0:34
5	In the *test* method, add an if statement to check if *number* is greater than 18.	0:32
6	Create a setter and getter for the field *age*.	0:55
7	Create a function named *sum* that receives two integer parameters and returns their sum.	1:32
8	Complete the factorial function.	2:08

Table 1: Experiment tasks and average completion times.

6 RELATED WORK

VoiceGrip [4] pioneered the development of an open-source system for voice programming, where users employ natural language with a specific syntax that is easy to grasp. This work as evolved to VoiceCode [5], which addressed various shortcomings of VoiceGrip, namely the need for dictating punctuation, navigation commands, and error correction. Notably, the system intelligently interprets abbreviations such as "*current record number*" as "*currRecNum*" and includes error correction mechanisms for instances where the system does not correctly recognise the spoken words.

SPEED [3] is an Eclipse plugin to support Spoken Java as a way to insert code into the editor. A user study with programmers [2] concluded that they were slower when compared to keyboard, results to which our experiment results are aligned. However, as speech recognition systems improve, we might achieve more effective solutions for programming by voice.

VocalIDE [14] is a specialized IDE focusing on voice commands. The development stemmed from a study gathering voice-based corrections for code snippets. VocalIDE facilitates the dictation of code word by word or letter by letter, and features context color editing, which allows users to select elements of the code by mentioning the color in which they are highlighted — these are automatically colored by the environment as the cursor moves.

VCL (Voice Command Language) [7] is a voice-to-code system where the same set of voice commands may generate Java, C, and Python. The study emphasizes the efficiency of combined voice and traditional input methods. However, specifics about code navigation remain unaddressed, a critical aspect in a system aiming to replace mouse and keyboard navigation.

VoiceJava [19] is a programming environment for Java designed to support voice commands. As with Jasay, VoiceJava performs transformations on JavaParser's ASTs, preserving code structure.

Voiceye [9] is a system that integrates voice, gaze, and mechanical switches for HTML and CSS coding. The study highlights the efficiency of voice commands over gaze-based inputs. Notably, the authors conclude that the inclusion of voice as a navigation tool is a valuable aspect, enhancing the overall usability of the editor.

7 DISCUSSION

Despite the encouraging early experiment results, we conclude that Jasay needs improvement regarding navigation commands and smarter context inference. We believe that our approach could be complemented with eye-gaze techniques (e.g., [16]) to provide efficient navigation. As another limitation, we only addressed the elementary constructs of Java, without providing support for constructs such as casts, annotations, and fine-grained editing of expressions.

AI code assistants recently became mainstream (e.g., GitHub Copilot[5]), while there are already numerous plugins for integration with popular IDEs (e.g., JetBrains IntelliJ, VS Code). Interaction with these systems is performed using textual descriptions of goals (e.g., what a function should return). With the advent of robust speech recognizers, such as Whisper, the voice modality for interacting with AI code assistants may become available soon. That will likely provide programmers with what our approach offers to a great extent, as well as many other capabilities of AI code assistants, despite not being based on projectional editing.

Despite that we targeted projectional editing, the core characteristics of our approach are not specific to it. The text phrases are translated to AST transformations, and the AST might be available regardless of using a projectional editor. For instance, IDE

[5] github.com/features/copilot

features for performing refactorings rely mostly in AST transformations. AST transformations are fragile when the code is not in a parseable state, while applying them in these cases may result in structurally unsafe modifications. Regarding the cursor location, determining its AST node every time it moves, requires intensive AST (re-)construction, which may not be at all times complete in the presence of syntax errors.

The main advantage of using a projectional editor is that the AST is always available without any parsing, while there are no syntax errors (malformed code). The AST evolves along with the editing process, while the cursor is always available as an AST node without further analysis. Our work explores this characteristic towards forming AST transformations from voice commands that preserve well-formed code.

ACKNOWLEDGMENTS

We thank the anonymous study participants. This work was partially supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT), under projects UIDB/04466/2020, UIDP/04466/2020, and UIDB/50021/2020.

REFERENCES

[1] Nurul Huda Baba and Dian Darina Indah Daruis. 2016. Repetitive strain injury (rsi) among computer users: a case study In telecommunication company. *Malaysian Journal of Public Health Medicine* 16 (2016). https://api.semanticscholar.org/CorpusID:207924911

[2] A. Begel and S.L. Graham. 2006. An Assessment of a Speech-Based Programming Environment. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. 116–120. https://doi.org/10.1109/VLHCC.2006.9

[3] Andrew Begel and Susan L. Graham. 2005. Spoken Programs. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC '05)*. IEEE Computer Society, USA, 99–106. https://doi.org/10.1109/VLHCC.2005.58

[4] A. Desilets. 2001. VoiceGrip: A Tool for Programming-by-Voice. *International Journal of Speech Technology* 4, 2 (2001), 103–116. https://doi.org/10.1023/A:1011323308477

[5] Alain Désilets, David C. Fox, and Stuart Norton. 2006. VoiceCode: An Innovative Speech Interface for Programming-by-Voice. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems (CHI EA '06)*. Association for Computing Machinery, New York, NY, USA, 239–242. https://doi.org/10.1145/1125451.1125502

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc.

[7] Sakib Hossain, Mabia Akter Emi, Mohsina Hossain Mishu, Raihana Zannat, and Ohidujjaman. 2021. Code Generator based on Voice Command for Multiple Programming Language. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. 01–05. https://doi.org/10.1109/ICCCNT51525.2021.9579880

[8] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 425–436. https://doi.org/10.1145/1993498.1993548

[9] Bharat Paudyal, Chris Creed, Maite Frutos-Pascual, and Ian Williams. 2020. Voiceye: A Multimodal Inclusive Development Environment. In *Proceedings of the 2020 ACM Designing Interactive Systems Conference (DIS '20)*. ACM. https://doi.org/10.1145/3357236.3395553

[10] Vaclav Pech, Alex Shatalin, and Markus Voelter. 2013. JetBrains MPS as a Tool for Extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13)*. Association for Computing Machinery, New York, NY, USA, 165–168. https://doi.org/10.1145/2500828.2500846

[11] Aidan Pine, Patrick Littell, Eric Joanis, David Huggins-Daines, Christopher Cox, Fineen Davis, Eddie Antonio Santos, Shankhalika Srikanth, Delasie Torkornoo, and Sabrina Yu. 2022. G_i2P_i Rule-based, index-preserving grapheme-to-phoneme transformations. In *Proceedings of the Fifth Workshop on the Use of Computational Methods in the Study of Endangered Languages*. Association for Computational Linguistics, Dublin, Ireland, 52–60. https://aclanthology.org/2022.computel-1.7

[12] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, and Ilya Sutskever. 2023. Robust Speech Recognition via Large-Scale Weak Supervision. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.), Vol. 202. PMLR, 28492–28518. https://proceedings.mlr.press/v202/radford23a.html

[13] Rianto, Arief Hermawan, and P. Insap Santosa. 2018. Knowledge and Prevention of Repetitive Strain Injury Among Computer Users. In *2018 International Conference on Orange Technologies (ICOT)*. 1–4. https://doi.org/10.1109/ICOT.2018.8705901

[14] Lucas Rosenblatt. 2017. VocalIDE: An IDE for Programming via Speech Recognition. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '17)*. Association for Computing Machinery, New York, NY, USA, 417–418. https://doi.org/10.1145/3132525.3134824

[15] André L. Santos. 2020. Javardise: A Structured Code Editor for Programming Pedagogy in Java. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming (Programming '20)*. Association for Computing Machinery, New York, NY, USA, 120–125. https://doi.org/10.1145/3397537.3397561

[16] André L. Santos. 2021. Javardeye: Gaze Input for Cursor Control in a Structured Editor. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming (Programming '21)*. Association for Computing Machinery, New York, NY, USA, 31–35. https://doi.org/10.1145/3464432.3464435

[17] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional Software. *SIGPLAN Not.* 41, 10 (oct 2006), 451–464. https://doi.org/10.1145/1167515.1167511

[18] Danny van Bruggen, Federico Tomassetti, Roger Howell, Malte Langkabel, Nicholas Smith, Artur Bosch, Malte Skoruppa, Cruz Maximilien, ThLeu, Panayiotis, Sebastian Kirsch (@skirsch79), Simon, Johann Beleites, Wim Tibackx, jean pierre L, André Rouél, edefazio, Daan Schipper, Mathiponds, Why you want to know, Ryan Beckett, ptitjes, kotari4u, Marvin Wyrich, Ricardo Morais, Maarten Coene, bresai, Implex1v, and Bernhard Haumacher. 2020. javaparser/javaparser: Release javaparser- parent-3.16.1. https://doi.org/10.5281/zenodo.3842713

[19] Tao Zan and Zhenjiang Hu. 2023. VoiceJava: A Syntax-Directed Voice Programming Language for Java. *Electronics* 12, 1 (2023). https://doi.org/10.3390/electronics12010250