Autocorrection in Projectional Editors

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

Ângelo Mendonça
amlfm@iscte-iul.pt
Instituto Universitário de Lisboa (ISCTE-IUL)
Lisboa, Portugal

ABSTRACT

People often mistype words when using keyboards. Word processors commonly feature autocorrection that checks for dictionary-based spelling mistakes and automatically performs text replacement after the user types a word. Programs are mostly described using text, and hence, the programmer may introduce typos when writing program identifiers (or keywords). In this paper, we describe an approach to integrate autocorrection in a projectional editor, capable of fixing program identifier typos. We implemented two modes of autocorrection as an extension of Javardise, one that resembles word processor autocorrection and a more experimental one based on the substitution of individual user keystrokes.

CCS CONCEPTS

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

KEYWORDS

Projectional editors, autocorrection, Java

ACM Reference Format:
André L. Santos and Ângelo Mendonça. 2024. Autocorrection in Projectional Editors. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming (‹Programming› Companion '24), March 11–15, 2024, Lund, Sweden.* ACM, New York, NY, USA, 5 pages.
https://doi.org/10.1145/3660829.3660844

1 INTRODUCTION

We know from personal experience that typing mistakes are frequent. However, we did not find any study that quantifies how frequent are these mistakes in programming, and how much they affect the performance and experience of code writing. However, a large-scale study on typing English sentences revealed that people make an average of 2.29 error corrections per sentence [2]. Although this error rate cannot be extrapolated to programming, it demonstrated that misspellings are relatively frequent. Given that program identifiers are words or abbreviations, typing them is also prone to mistakes.

In most popular IDEs, when a programmer mistypes an identifier by accident or because of incorrect spelling a compilation error is signaled with an error mark, which the programmer may visit and fix the error. Among other code transformation options,

‹Programming› Companion '24, March 11–15, 2024, Lund, Sweden

ACM ISBN 979-8-4007-0634-9/24/03
https://doi.org/10.1145/3660829.3660844

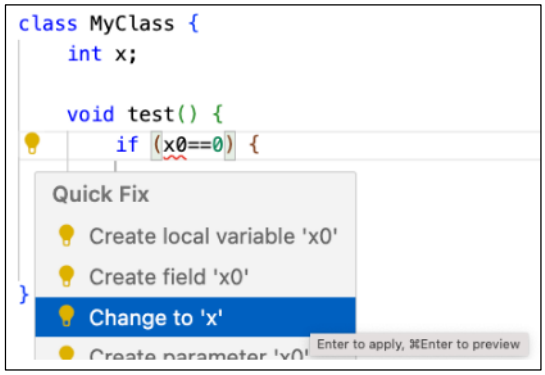

Figure 1: Quick-fixes for correcting identifiers in VS Code.

IDEs provide quick-fix actions to trigger text replacements with suggestions that match the identifiers in scope. Figure 1 presents an example of this sort of feature illustrated with VS Code[1]. Other industrial IDEs, such as IntelliJ IDEA[2] and Eclipse[3], offer similar features.

Quick fixes are useful but require additional interaction activity to trigger and select the actions. Our approach aims at fixing identifier misspellings, but in the manner that word processors do. The words (i.e., references to program identifiers) are corrected while the user is typing. We are not aware of code editors that provide this sort of feature.

A related feature dates back to LISP environments, namely PILOT, which provided a *Do-What-I-Mean* (DWIM) package [13] that performed corrections in program symbols during interpretation, based on string similarity. In contrast, our approach aims at a static time correct-as-you-type feature, eliminating the presence of transiently broken references that would otherwise require manual intervention.

We describe how we implemented autocorrection in a projectional editor through the injection of commands in addition to those of the programmer. The commands correct mistyped identifiers based on checking string similarity with identifiers in scope. These are expressed as any other editing command, and hence, an undesired correction may be canceled through the regular undo feature. Existing projectional/structured editors have not exploited the sort of autocorrection we propose [1, 4–6, 8–10, 12].

2 BACKGROUND

In projectional editors, the programmer sees a projection of a model that is at every moment well-structured, as opposed to raw text,

[1] https://code.visualstudio.com
[2] https://www.jetbrains.com/idea
[3] https://eclipse.org

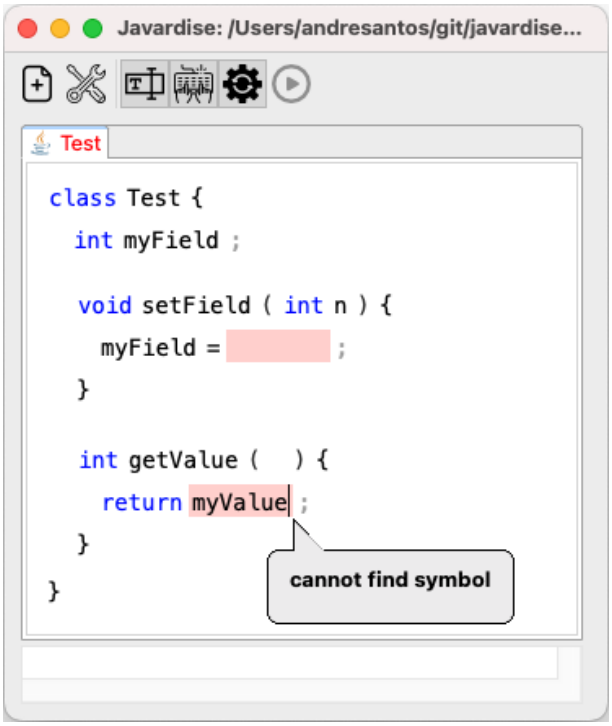

Figure 2: Screenshot of Javardise exhibiting compilation errors.

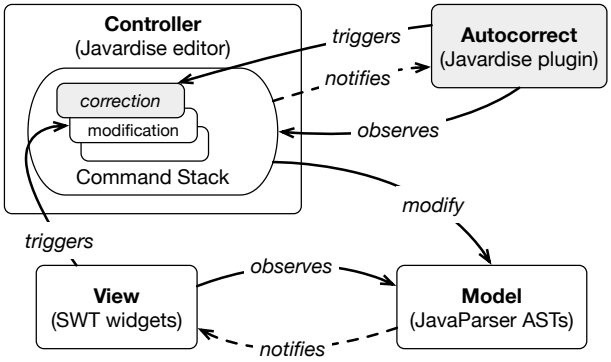

Figure 3: Model-view-controller architecture of Javardise and the integration of autocorrection (gray elements).

which may be in an unparseable state. Every program modification, such as adding a statement or renaming an identifier, is represented in a well-defined model transformation. In our approach, we exploit this characteristic of having the evolution of code represented in a sequence of discrete transformation commands, which one may analyze and manipulate.

Javardise [11] is a research projectional editor for Java, currently supporting a subset of the language. At first glance, the editor's appearance resembles a conventional one (see Figure 2). However, the user experience is that of a structured editor, characterized by not allowing malformed constructs. What is visible on-screen is a projection of an underlying model of the code that is at all times structurally correct (equivalent to being accepted by the language grammar). However, the code might have semantic errors (e.g., types) or missing expressions.

In contrast to tools such as Jetbrains MPS [10], there is no specific serialization format, Javardise works with regular Java files. JavaParser [14] is a library comprising classes that model Java code (ASTs) and a parser to instantiate those from source code. Javardise uses this library to load in-memory representations of the code. Upon modification, these are serialized back to Java code.

Javardise embodies a model-view-controller architecture (see Figure 3, white components). For each file, the *model* consists of a JavaParser's AST, whereas *views* are UI widgets built with Eclipse's SWT[4]. A view widget observes changes in the model and updates accordingly. However, it does not modify the model directly, instead, it triggers *commands* [3] whose execution is centralized in the *controller*. Commands have both the action and undo behaviors and are kept in a stack. When commands originate changes in the ASTs, observing views react accordingly.

3 IMPLEMENTATION

We implemented a proof-of-concept autocorrection feature by extending Javardise with a plugin that produces additional commands as corrections. The source code is available at the Javardise repository[5] under the subproject *autocorrect*.

Figure 3 (gray components) illustrates how the autocorrection plugin integrates with the base editor. It listens to the execution of commands and triggers a command that performs the necessary change whenever an opportunity for correction is detected. The additional autocorrection commands are treated by the editor as regular ones, and hence, one may perform undo to cancel their execution. We implemented two modes of autocorrection, which we detail next.

3.1 Command Refinement

One autocorrection mode resembles the feature of word processors and is based on command refinement. The commands that result from user interaction are "overridden" to better fit the existing code.

Autocorrection may occur whenever the user types an unresolved program identifier, either from scratch or by modifying an existing one. If applicable, an autocorrection command is generated and executed right after the mistake. Because we have a well-formed AST at every moment, we can reliably perform analyses immediately after every node insertion or modification.

We used JavaParser's symbol solver to attempt to resolve references of newly added identifiers. When resolution fails, we consider it an opportunity for autocorrection, triggering the match process. Figure 4 illustrates this autocorrection mode, where the user typed "myfield" (instead of "myField") followed by "=" to write an assignment, but the resulting instruction was "myField = " due to the injected autocorrection command.

In cases when the changes performed by a command do not lead to any broken references, we do not attempt any sort of correction.

[4] https://www.eclipse.org/swt
[5] https://github.com/andre-santos-pt/javardise

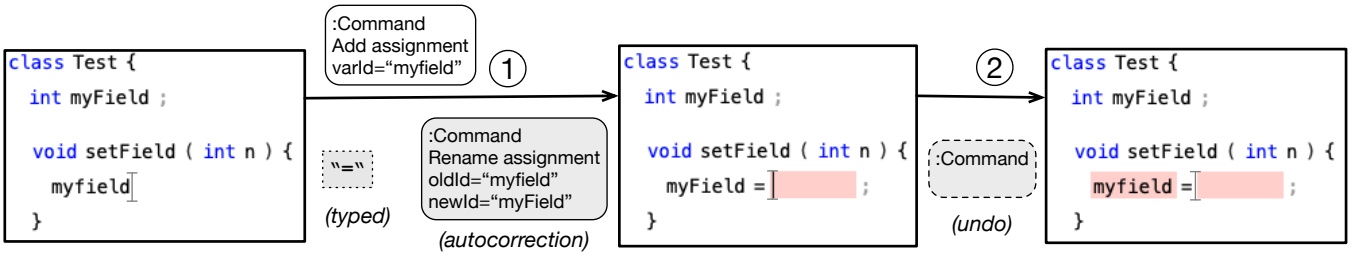

Figure 4: Autocorrection mode with command refinement. (1) a command is executed holding a broken reference, and a correction command is triggered; (2) undo is performed to reverse the correction effect.

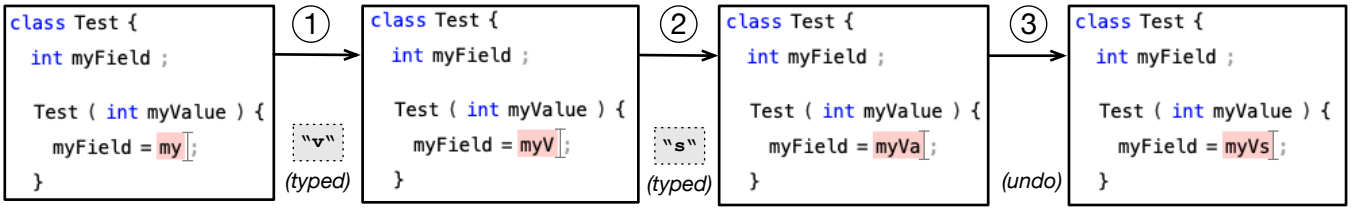

Figure 5: Autocorrection mode with keystroke substitution. (1) the user types "v" (lowercase), which is substituted by "V" (uppercase); (2) the user types "s", which is substituted by "a" (adjacent in keyboard); (3) undo is performed, and the previous keystroke is reclaimed.

On the other hand, the correction may be undesired for some reason, for instance, the user may wish to write yet-to-be-declared identifiers that are similar to existing ones. The user may cancel the autocorrection using the regular undo feature, rolling back to the state before the autocorrection command.

3.2 Keystroke Substitution

While typing an identifier, the user may notice a typo, and consequently, may hit backspaces and correct it as soon as it was noticed. In these situations, the user would not benefit from the previous autocorrection mode. Therefore, we implemented another, more invasive and experimental, mode of autocorrection based on keystroke substitution. In this mode, certain keystrokes may be replaced on the fly as the user types. For example, the user may type a lowercase "v" and observe on the screen an uppercase "V". Figure 5 illustrates how this mode works.

The implementation of this mode required more low-level interference with SWT events because every keystroke is an event of interest. The AST modification commands of Javardise do not work at this level of granularity, since a new command is triggered only when a modification (or insertion/deletion) is performed. In particular, if a variable expression is being edited, the command only takes effect once the focus is out of the widget.

Every time the focus changes to a text-editable widget holding an identifier, we attach a listener to capture the keystrokes therein (and detach once the focus is lost). Depending on which type of AST element the widget is addressing, we compute the valid identifiers. Every time a key is pressed, the current text content plus the new character is matched against the list of identifiers, which are trimmed to their prefix with a length equal to the current text. When matched, the prefix of the valid identifier replaces the text in the widget.

3.3 Matching Identifiers

Our main goal is to address all the programming elements that involve identifiers, except when these are part of declarations. It would not make sense to correct an identifier that does not yet exist. In these cases, regular spell checkers may come into action, a feature nevertheless already available in modern IDEs.

Corrections are based on calculating the string distance (Levenshtein edit distance [7]) to valid identifiers in scope, taking into account the type of element that is being referred. A threshold value can be altered to define how closely strings should match to trigger the autocorrection. Next, we detail the different contexts in which an identifier correction may be performed.

Type references. These apply to declarations of fields, methods, parameters, and local variables. Except for type declarations, every other declaration refers to an existing type. Autocorrection will try to match the set of existing project types, as well as the types imported in the file that is being edited. This could be broadened to all the types in the classpath.

Variable expression. These apply to expressions that refer to class fields or local variables (including parameters). Autocorrection will try to match a valid variable identifier according to the scope hierarchy of name resolution: local variables followed by fields.

Field expression. These apply to field access expressions (dot notation), where a field is accessed by name (right) on an expression scope (left). Autocorrection will consider the type of the expression to find a valid field identifier within that scope. If the scope corresponds to a type name, instead of a variable, a match to static fields of that type will be performed.

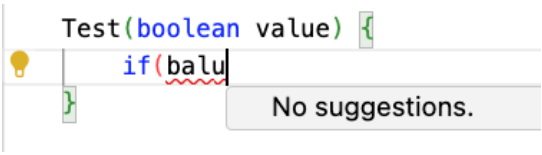

Figure 6: Autocompletion in VS Code of a partially mistyped identifier (an almost valid identifier suffix with one mistyped character).

Method expression. These apply to expressions and expression statements that hold method calls. Autocorrection will try to match a valid method identifier in scope. Method calls may have an expression scope; in that case, the behavior is analogous to field expressions.

4 DISCUSSION

The availability of good code autocompletion features and their intensive use may imply that the autocorrection feature we propose is marginally useful since barely any identifier spellings would occur. However, a user may still mistype characters, and autocompletion is not as useful for dealing with text that was already (mis-)typed. Figure 6 presents an example in VSCode illustrating this situation.

The sort of autocorrection feature we propose could also be implemented in textual code editors. However, the editing process's nature could complicate the implementation, given that editing actions are fuzzy (that is, without precise boundaries), hence more difficult to track reliably. Furthermore, it requires dealing with code that is not fully parseable due to malformedness.

The main advantage of a projectional editor is that every editing step is well-defined in a command, turning the evolution of the source code into a "discrete" process. In addition, the code elements are represented in dedicated widgets aware of the AST node being edited. This allows for precise lookups of identifiers to perform the correction.

When compared to autocorrection of regular text (as in word processors), achieving the analogous behavior with source code is a considerably simpler problem. In word processors, corrections are based on dictionary lookups to check word similarity, but may also use the context of the word (immediate previous words) to choose a substitution that maximizes likelihood with probabilistic models. However, the state space of natural languages is wide and context has ambiguity. In programs, the set of identifiers that are valid in a given scope is both well-defined (no ambiguity) and with relatively low cardinality (say, often not reaching hundreds). This characteristic facilitates the accuracy of the autocorrections.

We did not investigate how users perceive the sort of autocorrection feature we proposed. We speculate that users would find the command refinement autocorrection mode familiar, as it resembles the autocorrection feature that has been available in word processors for many years. The autocorrection mode based on keystroke substitution is more invasive and we acknowledge that it may confuse users. Further, we did not devote much time investigating less obvious undesired behaviors of such an autocorrection feature. This aspect requires further investigation and user experiments.

An obvious drawback of autocorrection is when undesired modifications are performed. However, given that those can be canceled with *undo* — a simple and already existing mainstream command — it should not compromise usability significantly. Recall that our autocorrection feature never modifies an identifier that is not broken. Hence, we foresee two main undesired scenarios: (a) the identifier chosen by the autocorrection is not the desired one, and (b) autocorrection overrides the deliberate intention of the user to type an identifier that does not yet exist. Both scenarios are triggered due to the presence of similar identifiers in scope. Regarding the first scenario, the user overcomes the situation by performing undo and writing the desired identifier. Without autocorrection on, the user would have anyways to go back to the identifier location to correct it. Regarding the second scenario, the user only has to perform undo and continue writing the code.

In addition to the evaluation of the usability of our approach, a user study involving code writing and modification tasks could measure how often autocorrections are performed, as well as how often these are canceled. The ratio of autocorrections per user command would measure how useful the feature is, whereas the ratio of canceled autocorrections per autocorrection command would measure its accuracy. Accurate autocorrections save time and typing effort for the user and in this way have the potential to make the programming activity more efficient and ergonomic (less physical activity).

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their valuable suggestions to improve this paper. This work was partially supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020].

REFERENCES

[1] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 595, 16 pages. https://doi.org/10.1145/3544548.3580785

[2] Vivek Dhakal, Anna Maria Feit, Per Ola Kristensson, and Antti Oulasvirta. 2018. Observations on Typing from 136 Million Keystrokes. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3174220

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

[4] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: a lightweight user interface for structured editing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 654–664. https://doi.org/10.1145/3180155.3180165

[5] Amy 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. https://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. https://doi.org/10.1145/3287324.3287462

[7] Vladimir Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966).

[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] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (jan 2019), 32 pages. https://doi.org/10.1145/3290327

[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] 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

[12] 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

[13] Warren Teitelman. 1969. Toward a Programming Laboratory. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, Donald E. Walker and Lewis M. Norton (Eds.). William Kaufmann, 1–8.

[14] 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

Received 2024-02-08; accepted 2024-02-26