# A Framework for a Programmer's Minion

## EXTENDED ABSTRACT

Steven P. Reiss, Qi Xin
Department of Computer Science, Brown University
Providence, RI. 02912 USA
{spr,qx5}@cs.brown.edu

***Abstract*—Programming environments should help the programmer. Today's environments do a lot along these lines, yet more is possible. We are developing facilities within the Code Bubbles environment to proactively assist the programmer. These facilities attempt to take care of mundane tasks that the programmer otherwise would need to do, effectively acting as a programmer's minion. This paper describes the framework provided by the environment to support these tasks and an evaluation of the effectiveness of the initial minion implementation.**

***Keywords—automatic correction, programming environments, programming tools.***

### I. Motivation

Programming environments should assist the programmer as much as possible. Today's environments, e.g. Eclipse, Visual Studio, IntelliJ, and XCode, offer many features designed to make programming easier. Yet more assistance is possible. Our goal is to have the environment effectively act as a programmer's minion, taking care of relatively mundane tasks and fixing obvious things as the programmer works. Such tasks might include fixing typos, adding import statements, fixing simple semantic errors, etc.

To this end, we are building a framework within the Code Bubbles programming environment [1,8] to support a wide variety of automatic fixes. The goal of this framework and its included fixes is to lessen the load on the programmer. As such any changes made by the environment acting as a minion must meet three requirements:

- First, the changes should be *unobtrusive*. The environment should not correct things while the user is likely to make changes that would affect them. It should not require the user to change focus from their current position to a previous error.
- Second, any changes made must be *correct*. The environment should ensure that changes are the right changes. The changes should actually fix a problem, not introduce new problems, and be unambiguous.
- Third, any changes should be made *the way that the programmer would do it.* The programmer should not have to go back and correct or reedit an automatic change because it wasn't done their preferred way.

### II. Overview and Contributions

Supporting general automatic correction requires a support framework. There are several problems that such a framework needs to address:

- First, the framework must provide a consistent way of detecting when to make a correction. Our framework maintains a current edit region, the portion of text where the user is editing. It tracks the cursor in that region as well as all error messages. Changes are only made to an active edit region and are made based on the current error and warning messages associated with that region.
- Second, the framework must be able to generate fixes for the correction. We introduce the notion of adapters, each of which is responsible for a certain type of fix. The adapters are passed an error message and generate prioritized possible fixes for that error. For example, a spelling adapter would generate possible corrections for an undefined identifier. Priorities are used to distinguish more likely fixes from less likely ones and to identify when fixes might be ambiguous.
- Third, the framework must validate the fixes. We provide facilities for creating a private copy of the source, making the fix in that copy, and then checking the errors that result. This is done in a background thread. A fix is only made if it removes the original error, does not introduce any new errors at the point of the fix, reduces the number of errors, and is unambiguous at its priority level. Additional checks ensure that the original error is still present in the source and that the current cursor is not located within the fix.
- Finally, the framework must make the fix. This involves rechecking that the fix is still appropriate.

### III. Adapters

The initial implementation of our minion framework includes a suite of adapters aimed at fixing common mistakes. These include:

**Spelling Correction.** The spelling adapter is triggered by an error that covers a single identifier or a single identifier followed by a single character operator followed by another identifier. It uses auto-completion [5] results along with the set of keywords and available types as initial candidates, finds candidates whose edit distances are small, and then prioritizes candidates by edit distance. The adapter includes a user-editable list of exceptions where no correction should be made to handle the cases where it may make mistakes.

**Syntax Corrector.** The syntax adapter looks for syntax errors that tell what token needs to be inserted, deleted, or replaced. It sets up a candidate based on the error message and checks if the proposed fix is valid before making the change. It generally only works once the user has moved on to the subsequent line.

| Adapter | # Uses / Editing hour | Manual Fix Time (sec) | Time Saved / Editing Hour (sec) | % Saved | Mean # Alternatives | Std. Dev. |
|---|---|---|---|---|---|---|
| Spelling | 14.55 | 5.30 | 77.115 | 2.14% | 1.22 | 0.54 |
| Syntax | 3.21 | 5.30 | 17.013 | 0.47% | | |
| Imports | 6.96 | 10 | 69.600 | 1.93% | 2.09 | 3.16 |
| Quotes | 0.16 | 5.30 | 4.452 | 0.12% | | |
| **TOTAL** | | | **168.180** | **4.67%** | | |

**Figure 1. The results of the log studies. We measured how many times each type of adapter generated a fix per hour of editing. The estimate of time to do a simple fix was obtained from the log studies, the time for fixing an import is a conservative estimate. The last two columns show the mean and standard deviation of the number of alternatives that were considered. The other adapters were not used a significant number of times.**

**Import Corrector.** The import correct checks for errors indicating an undefined type (or an undefined identifier that starts with an upper case letter). It finds all the viable candidate types and then prioritizes them according to a model of how relevant the type is likely to be to the application based on whether the type is part of the user's code and whether it has been imported in another file either directly or indirectly. It tries the to find a validated candidate at the highest possible priority that is not ambiguous. Eclipse's smart import capability is then used to ensure the import is added as the programmer would want it to be.

**Quote Corrector.** The quote correction adapter looks for errors related to missing or extra quotation marks. If it finds an unclosed quotation on a previous line, it finds locations on that line that would be valid start or end points for a string (e.g. operators, parenthesis). It generates a candidate fix by inserting a quotation mark before that character and checking if the results compiles correctly. All possible insertions are assigned the same priority.

**Return Correction.** The return corrector adds a default return statement to a new method that requires one. Even if the result statement needs to be replaced later in the editing process, this can be helpful as it removes error messages while typing and cleans up the Code Bubbles display.

**Try-Catch Corrector.** The try-catch corrector looks for try statements where the user has added code that throws an uncaught exception. It then adds an empty catch clause to the try statement for that exception.

## IV. Experience and Evaluation

To analyze the effectiveness of the various adapters and the overall framework, we analyzed two months of command logs from the Code Bubbles environment. The Code Bubbles environment records anonymous information on its usage based on a user opt-in strategy. We considered the recorded data about the sequence of commands issued by the user.

We first analyzed the data to determine when the programmer was actively editing (i.e. not debugging, thinking, or simply looking at code). Then we looked at how often each of the adapters was used per hour of editing time. We also analyzed the logs to determine how much time was spent to make a simple fix, using the median value (5.30 seconds) as the base line. From this we generated the results shown in Figure 1. These results show the minion framework saves about 4-5% of the programmer's editing time.

These estimates are just that, estimates. The threats to their validity include that the users over the 2 month period might not be typical and that the estimation of time to fix the problems could be wrong.

## V. Related Work

Our work was motivated by many existing spell-checking tools such as Microsoft Word and auto-completion techniques like tab-completion and Google Search Suggest.

Several projects have extended an integrated development environment's auto-completion functionality. Our system is related to [3] where the auto-completion is designed to be error-tolerable. Given a user-typed prefix as the query, which might be erroneous, the auto-completion system would suggest a list of extension candidates whose prefixes are the closest to the query. Like us, they use edit distance as the measure. [7] implemented auto-completion for partial expressions. For an API method with missing parts, the algorithm first generates candidates for completion based on the type information. Active code completion [6] replaces the simple auto-completion menu with user-interactive interface for completion enhancement. [9] combines software changes information to code completion for improvement. [2] performs repository code search to find similar method usage patterns for effective code completion. Keyword programming implements auto-completion based on keywords [4]. It generates code containing the keywords while fitting the context according to the Java type system. The newer versions of Eclipse do automatic import insertion, but only if the type is unique. They do not use a priority model as we do and hence miss a lot of cases.

## VI. Conclusions

Code Bubbles, including the correction facilities, is available for download from http://www.cs.brown.edu/people/spr/codebubbles. The current implementation of the framework is included in the Code Bubbles source available through SourceForge. The learning package, log analysis tools, log data used, and open source projects that were analyzer are available upon request.


### Acknowledgment

This work was supported by the National Science Foundation grant CCF1130822.

## References


1. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," *ACM/IEEE International Conference on Software Engineering 2010*, pp. 455-464 (2010).
2. Marcel Bruch, Martin Monperrus, and Mira Mezini, "Learning from examples to improve code completion systems," *Proceedings of ESEC/FSE 2009*, pp. 213-222 (2009).
3. Surajit Chaudhuri and Raghav Kaushik, "Extending autocompletion to tolerate errors," *Proceedings of SIGMOD 2009*, pp. 707-718 (2009).
4. Greg Little and Robert C. Miller, "Keyword programming in Java," *Proceedings ASE 2007*, pp. 84-93 (November 2007).
5. G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?," *IEEE Software* **23**(4) pp. 76-83 (2006).
6. Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers, "Active code completion," *Proceedings of the International Conference on Software Engineering 2012*, pp. 859-869 (2012).
7. Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman, "Type-directed completion of partial expressions," *Proceedings of PLDI 2012*, pp. 275-286 (2015).
8. Steven P. Reiss, Jared N. Bott, and Joseph J. La Viola, Jr., "Plugging in and into Code Bubbles: the Code Bubbles architecture," *Software Practice and Experience*, (2013).
9. Romain Robbes and Michele Lanza, "Improving code completion with program history," *Journal of Automated Software Engineering* **17**(2) pp. 181-212 (June 2010).