Quick Fixes for novice programmers: effective but under-utilised

Neil C. C. Brown
King's College London
London, UK
neil.c.c.brown@kcl.ac.uk

Jamie Ford
King's College London
London, UK
jamie-ford@outlook.com

Pierre Weill-Tessier
King's College London
London, UK
pierre.weill-tessier@kcl.ac.uk

Michael Kölling
King's College London
London, UK
michael.kolling@kcl.ac.uk

ABSTRACT

Professional software development environments typically provide "quick fixes" for common program errors: a common solution to an error that can be enacted with a single click. For example, an "unknown type" error can be fixed by adding an import for the type, or an "unknown variable" error can be fixed by changing a misspelt identifier to match the name of an already-declared variable. The BlueJ environment recently added some support for quick fixes, and in this paper we use the associated Blackbox dataset to investigate how they are used by novice programmers. We use a combination of automatic filtering of over 100 000 quick fixes, and manual categorisation of 900 programming session fragments. We find that acceptance of suggestions ranges from 1–17% for different quick fix types, usually within 3–5 seconds, and many manually performed alternatives are similar to the actions of unselected quick fixes. Furthermore, we find that users are faster and are more able to make productive progress when quick fix support is available, especially for fixing imports. This data can be used to inform design of future quick fix systems. There are pedagogical arguments for and against providing such a feature to novice programmers and we provide some initial discussion on the matter.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; • **General and reference** → Empirical studies.

KEYWORDS

Quick fixes, Blackbox, Programming education

ACM Reference Format:
Neil C. C. Brown, Jamie Ford, Pierre Weill-Tessier, and Michael Kölling. 2023. Quick Fixes for novice programmers: effective but under-utilised. In *The United Kingdom and Ireland Computing Education Research (UKICER) conference (UKICER 2023), September 07–08, 2023, Swansea, Wales Uk.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3610969.3611117

UKICER 2023, September 07–08, 2023, Swansea, Wales Uk

ACM ISBN 979-8-4007-0876-3/23/09…$15.00
https://doi.org/10.1145/3610969.3611117

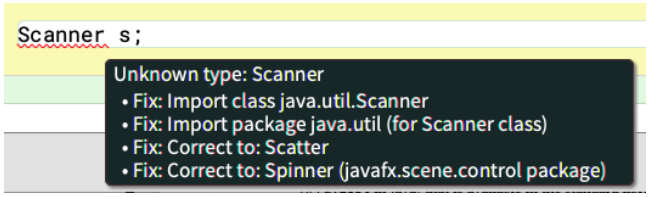

Figure 1: A screenshot of a compiler error with four suggested clickable quick fixes in BlueJ's Java editor.

1 INTRODUCTION

Programmers of all levels of ability encounter errors in their programs. For compiled languages, this usually comes in the form of compiler errors. Some of these compiler errors are complex errors that require much thought to solve. However, many of these errors result from trivial mistakes that are made by many programmers, and often they have an obvious fix. For example:

- An "unknown type" error that is encountered for a type that is available in the standard library (such as `ArrayList` in Java) may be fixed by importing a class or a package.
- Modifying a variable name to match a similarly-named declaration (e.g. "totl" to "total") is a common way to fix "unknown identifier" errors when there is a similar name already declared.
- Adding a surrounding try/catch statement is a common way to fix an "uncaught exception" error.

Many of these fixes are likely to be useful to the programmer and they require a relatively trivial modification of the program's source code. In these cases, professional development environments add *quick fixes*: usually shown as a popup window next to the error, they offer a list of around 1–3 fixes, which modify the program source to implement the fix (see Figure 1 for an example). Thus, the error can be very quickly solved, and productivity – usually the key metric in professional environments – can be increased.

For novices, there is clearly a different trade-off. We want novices to learn. In the extreme case, if there was a button for "write my whole program" (perhaps via AI) this would be ideal for experts and worthless for novices: very productive but no learning would ensue. In the more modest case of quick fixes this trade-off is still present: do quick fixes prevent novice programmers getting stuck on trivial errors and thus increase learning overall, or do they rob novices of valuable learning experiences, thus keeping them in a state of shallow understanding? To explore this question, we first need data about how novices use quick fixes when they are available.

Quick fixes were recently added to the BlueJ environment's Java editor (see Figure 1). Their usage was subsequently recorded in the Blackbox data set [3], providing a useful opportunity to observe novice usage in the wild. To develop an initial understanding of how novices use this feature, our research questions (RQs) are:

- RQ1: Which quick fixes were most frequently presented to users, and which were most frequently used?
- RQ2: How long did users take to select a quick fix from the presented list – is it a quick decision or a long consideration?
- RQ3: If users did not enact a quick fix, did they modify the line themselves in a way that could have used the quick fix?
- RQ4: How do novices act differently when quick fix support is entirely unavailable, versus when it is available (regardless of whether they use them or not)?

2 RELATED WORK

Compiler error messages are a longstanding area of interest in computing education. Becker et al. [2] surveyed the extensive work that has been done on investigating and improving the error messages. However, to our knowledge there has been relatively little investigation of quick fixes. We define quick fixes as: a suggestion shown alongside a compiler error message that can be executed to modify the user's code with a probable fix for the error, which are offered without knowledge of what the user is trying to achieve. Thus we are not interested in next-step hints (which involve knowledge of the user's aim), or general code quality suggestions (which are not in response to an error), or hints that cannot be automatically executed such as in the work by Marwan et al. [8].

Phothilimthana and Sridhara [10] investigated students requesting hints with compiler error messages, but the hints could not be automatically executed. Hartmann et al. [5] investigated showing potential code fixes to students, but they judged the usefulness of the fixes manually rather than looking at whether users chose to enact them (which was only a partially-supported feature). Ahmed et al. [1] investigated student performance with tools to help fix syntax errors, but they investigated overall performance when the tool was available, without specifically looking at executions of fixes with the tools. They found that quick fixes did help students resolve errors more quickly when available.

We believe this paper is the first detailed large-scale investigation of student use of automatically-executable quick fixes for compiler errors, which investigates how and when the fixes are used.

3 DATA

BlueJ is an IDE for Java and Stride primarily used by novices, with a median age of 16. Of their users, 70% are in secondary education and 64% are using it for their first programming experience [4]. The Blackbox dataset has been collecting data from opted-in BlueJ users since 2013 [3]. The strength of the Blackbox dataset is its global reach and large size, albeit with a weakness that all users are anonymous and with no information on demographics or their intended activity, etc. For the current observational study, we do not believe that this reduces its usefulness.

Quick fixes were added to BlueJ in version 5.0.0 in January 2021, but they were not recorded into Blackbox until version 5.0.3, released 28 March 2022. In this study we will use the data from BlueJ 5.0.3, up to a cut-off of 23:59:59 UTC, 31 December 2022, looking at quick fixes in Java programs. The study was carried out in accordance with the research ethics process at our institution.

The following four sections (section 4 to section 7) explain the method and results for each of our four research questions in turn.

Table 1: The number of times each fix action was offered and executed. Multiple fixes can be offered for the same error. Some fixes are always offered together (e.g. solving an unknown class by importing the class or whole package) hence there are some duplicate offered counts. The final column is the executed count divided by offered count, to give relative rates of take-up. These figures are low mainly because users often do not use any available quick fixes, but also because at most one fix can be chosen when multiple are offered. Therefore, 100% is possible for a given row, but not for the *all actions* row, even if the users always selected a fix. These fixes were shown across 1 842 411 errors, so the maximum overall take-up for *all actions* would be 1 842 411 / 2 908 694 = 63.342%.

Quick fix action	Number / executed	Number = offered	Percentage executed
Spelling/typo correction	94 870 /	1 194 216 =	7.944%
Add class import	41 494 /	238 123 =	17.425%
Add local variable declaration	13 305 /	574 659 =	2.315%
Add package import	6 401 /	238 123 =	2.688%
Add field declaration	5 245 /	574 659 =	0.913%
Replace = with ==	3 893 /	60 261 =	6.460%
Add throws declaration	2 514 /	16 654 =	15.095%
Add try/catch statement	1 400 /	11 999 =	11.668%
All actions	169 122 /	2 908 694 =	5.814%

Table 2: The Damerau-Levenshtein case-insensitive edit distances of the spelling-correction fixes that were offered to the user, and that were chosen by the user to be executed. Zero distance is possible if the only difference was the case. Note that some errors may result in multiple fixes being shown, and thus the last column could not be 100% even if a fix was always chosen, because users can only select at most one offered fix. The total number of errors with an offered spelling correction fix was 522 439, therefore there were on average 783 009/522 439 = 1.499 offered spelling correction fixes whenever there was at least one.

Edit distance	Number / executed	Number = offered	Percentage executed
0	11 060 /	106 104 =	10.424%
1	20 231 /	460 743 =	4.391%
2	6 687 /	216 162 =	3.094%
Total	37 978 /	783 009 =	4.850%

Figure 2: The proportional frequency (within each quick fix action) of the time taken by the user to click to execute the fix. Times beyond 30 seconds are excluded.

4 FREQUENCIES (RQ1)

We queried the text for each quick fix from Blackbox for the specified time period, and we categorised them according to their action. (For example, "Add import for java.util.ArrayList" and "Add import for java.awt.Window" perform the same action: add a class import.) Then we used the counts of each across all the data.

There are several different Java compiler errors that can trigger BlueJ quick fixes. The overwhelming majority of errors (over 95%) that have a quick fix offer are "cannot find symbol". This error indicates that a particular identifier in the program does not match a previously declared or imported name.

The error message does not uniquely determine the available quick fixes, however, as an unknown identifier could be fixed by adding an import or correcting a typo. The frequencies of different *offered* quick fixes are shown in Table 1. Correcting spelling of an identifier to a similar identifier (e.g. "Peice" to "Piece") is the most frequently shown kind of fix, followed by adding a declaration of a missing variable. Table 1 also shows frequencies of *executed* quick fixes, including relative rates. For example, where a user sees an offer to fix a spelling correction, they click it 7.944% of the time.

The spelling/typo correction fix compares the lower-case version of the found unknown identifier to other declared identifiers in the code. The edit distance is calculated using the Damerau-Levenshtein algorithm (with cost 1 for delete, insert, replace and swap). Any identifier with an edit distance ≤ 2 is suggested. The edit distances of the chosen identifiers are shown in Table 2. An edit distance of zero is possible in this calculation where the two identifiers only differed by case, suggesting this is often the problem, and these instances are where users are most likely to execute the fix.

5 TIMINGS (RQ2)

For calculating the time taken, we use the time between the fix being shown and the fix being executed. Note that Blackbox has both a source time (on the originating computer) and a server time (when the event is recorded on the Blackbox server). It is important that we use the source time because there can be an unpredictable lag between the event being generated on the originating computer and being recorded into the database. For timings that may only be a few seconds it is important to use the source time.

Table 3: The average time taken for the user to click the fix to execute (non-executed fixes are excluded) for each kind of quick fix action. Times greater than 30 seconds are excluded. The bottom row has the average across all actions. The use of the average rather than median is explained in Section 5.

Quick fix action	Average time (seconds)
Correct spelling/typo	3.772
Add variable/field declaration	4.689
Add class/package import	3.655
Replace = with ==	3.534
Add throws declaration	4.662
Add try/catch statement	5.289
All actions	3.847

The timing of the quick fixes being shown and being executed are recorded using wall-clock time, to the second, so the difference between the two times is an integer number of seconds (and can be zero, in some cases where the user clicked in under a second). Thus the data is more granular than we might like, given that the mode (most common value) of the time to click is one second for all fixes.

We performed some preprocessing of the times after looking at the data. All of the times have a very long tail (up to several days long) so we only look at the times up to a maximum of 30 seconds, which excludes the 3.5% of the data values in the long tail beyond that. As is shown in Figure 2, the timings of all of the fixes is quite skewed towards the low end and there is no major difference between any of the quick fix actions in this regard.

The average time (after the values greater than 30 seconds have been excluded) is shown in Table 3. Although the average is not usually recommended on skewed distributions, the mode is identical for each and on this coarse integer data the median is uninformative (it is either 2 or 3 for all), therefore the average is the most informative summary measure here.

One possible concern is that users may be mindlessly selecting fixes without actually reading them, so we checked if the time to select differs significantly by category (which would indicate the users do differentiate between the fixes). A Kruskal-Wallis test confirms that the effect of fix-action is statistically significant at $\alpha = 0.05$ ($H(5) = 3418.7$, $p < 0.001$), although this is to be expected on such a large sample, because statistical tests are significant even for small differences when sample sizes are very large [6, 7].

6 MANUAL FIXES (RQ3)

Given that the rate of quick fix selection is relatively low, an obvious question is: what are users doing if they don't choose the quick fix? This is a difficult analysis to perform automatically as there may be quite a variety of code editing behaviours. Thus we analysed this manually: we randomly sampled 100 cases for the three most frequently used quick fix actions where the fix was shown but *not*

Table 4: The actions taken by the user after a quick fix was shown, but no fixes were executed, for 100 randomly sampled instances of each of (a) an import class/package quick fix, (b) a spelling/typo quick fix, and (c) a variable/field declaration quick fix. The time is the median time between the error and the action taking place.

	Action	Count	Time (secs)
(a)	Added class import manually	44	25.5
	Added package import manually	22	13.0
	Removed line with error	14	34.5
	Fixed existing import	9	37.5
	Edited type usage	1	119.0
	Unavailable	1	
	Did not fix during session	9	
(b)	Corrected name	54	5.0
	Declared variable	11	29.0
	Removed line with error	10	37.0
	Imported type	9	79.5
	Changed line another way	7	20.0
	Declared class	2	484.0
	Did not fix during session	7	
(c)	Renamed to match existing variable	33	6.5
	Declared variable on line	19	3.0
	Removed line	15	12.0
	Declared local variable before line	9	39.0
	Declared field	7	43.0
	Changed line another way	6	14.0
	Renamed existing declaration	4	25.0
	Did not fix during session	7	

executed. We then analysed the editing behaviour afterwards on that line in order to classify the subsequent user actions.

Table 4 part (a) shows the results of this process for add-import quick fixes. It can be seen that in two-thirds of cases the users performed the suggested quick fix edit manually, but more slowly than the average time to select the quick fix (3.655 seconds). Part of the reason for the slow time is that users did not necessarily fix the error immediately: sometimes they performed other edits and then went back to add the necessary imports.

Table 4 part (b) shows the results of this process for spelling/typo correction quick fixes. It can be seen that in 54 of the 100 cases the users did correct the identifier (and on further examination, 48 of these 54 were to items within an edit distance of 2, so to items that would have been shown in the quick fixes). This was done fairly quickly, with a median of 5.0 seconds compared to the 3.772 average time to select such a quick fix.

Table 4 part (c) shows the results of this process for variable declaration quick fixes. In this case, the line like "`foo = 2`" for an unknown variable foo is changed to "`_type_ foo = 2`" with the type placeholder selected so the user can immediately enter the type. Users were actually faster manually adding the type (median 3 seconds) than using the quick fix and then writing the type (mean of 4.7 seconds). In 26 cases the manual fix was one of the suggested fixes (19 for local variable, 7 for field), but the majority had alternative fixes: 33 cases instead renamed the variable usage, 4 renamed a declaration, and 9 declared a local variable at some other location.

7 WITH VS WITHOUT (RQ4)

The manual alternatives to fixes are not the complete story. They do not tell us whether the act of merely seeing the fix helped users, or what the users who did execute a fix would have done had the automatic fix not been available. For this, we need to compare what users do when quick fixes are available, to what they do when quick fixes are not available at all. Thankfully, this data is available in the form of a natural experiment: before and after quick fixes were added to BlueJ. We can compare the actions of users when quick fixes are not available, to when quick fixes are available.

There are two possible baselines for the unavailable case, each with a potential confound. We could pick users from the equivalent time period in Autumn 2020 before quick fixes were added to BlueJ. This was during the COVID-19 pandemic when classes were likely to be virtual, and is two years prior. Previous research [4] has shown that seasonality is a major factor so choosing the same time of year is important, but long-term trends are relatively slow (and no obvious effect of the pandemic on usage was found [4], so choosing 2020 should not be a problem). Another alternative is to pick users in the same time period in 2022 from the 18% of users who have not upgraded their BlueJ version to BlueJ 5. We do not know if this is for a reason, however: maybe they are more likely to be at institutions with less IT support, maybe they are longer-lasting users who have been using BlueJ for years. Ultimately, we processed both these options rather than picking one. We compared (a) users from the same annual time period (October and November) in 2020 vs (b) a set from 2022 without quick fix support vs (c) a set in 2022 with quick fix support. To ensure we were comparing like-for-like activity, we picked out occurrences of the compiler error "unknown symbol - class Scanner" (the most frequently unimported type that quick fixes are shown for) to see what users did next, and did the same for the compiler error "cannot not find symbol - name" (where the "`name`" part was allowed to vary in case; confusingly, the actual name "`name`" was the most frequently occurring variable name found among the quick fix suggestions).

Table 5 shows the outcome of tagging such instances without quick fix support available in 2020 and in 2022, to when it was available in 2022, for the unknown type `Scanner`. It seems that users were much more able to fix import errors with quick fix support available: 76 out of 100 ended up with a correctly imported type with quick fix support available, compared to 43 out of 100 and 61 out out of 100 without quick fix support. These actions were also performed much faster when quick fix support was available.

Table 6 shows the analogous outcome for the unknown identifier "`name`". Here, the benefit of the quick fixes is less clear. With quick fix support users were more likely to fix lower/upper case errors or add a declaration in place, but this seems to be instead of other successful fixes rather than not fixing: 82 out of 100 fixed the error with quick fix support, in contrast to 81 out of 100 and 83 out of 100 in the baselines. This will have been affected by our choice of "name" as the erroneous identifier; had we picked "lenght" (the most popular obvious typo) the fixes would have been composed much more of

Table 5: In response to an error about the unknown type `Scanner`, the counts of actions taken in 100 instance *without* quick fix support in 2020 and 2022, compared to 100 other instances *with* it available. The actions above the middle line are those that successfully resulted in an imported type. The times are median times between the error and the action taking place.

	Count			Time (seconds)		
Fix	Without (2020)	Without (2022)	With	Without (2020)	Without (2022)	With
Added class import	26	22	50	63.0	27.0	11.5
Added package import	9	26	13	84.0	36.0	25.0
Edited existing import	8	13	11	21.5	14.0	15.0
Edited type usage	0	0	1			3.0
Fully qualified usage	0	0	1			1151.0
Removed line	25	23	13	23.5	3.0	3.0
Otherwise edited line	6	0	2	40		123.5
Unavailable	1	0	1			
Did not fix during session	24	16	8			

Table 6: In response to an error about the unknown identifier "`name`", the counts of actions taken in 100 instance *without* quick fix support in 2020 and 2022, compared to 100 other instances *with* it available. The actions above the middle line are those that successfully resulted in an imported type. The times are median times between the error and the action taking place.

	Count			Time (seconds)		
Fix	Without (2020)	Without (2022)	With	Without (2020)	Without (2022)	With
Changed to another name	38	40	28	5.0	9.5	12.0
Case changed (e.g. name to Name)	7	7	20	3.0	4.0	5.0
Declaration added on same line	3	7	15	33.0	80.0	15.5
Local declaration added on earlier line	7	5	6	68.5	33.0	16.0
Field declaration added	9	4	7	32.0	69.0	30.0
Other fix	9	15	3	27.0	48.0	7.0
Removed line	12	11	11	12.0	6.0	24.0
Did not fix during session	9	6	7			

changing to another name. Out of the name changes, most were to items beyond edit distance of 2, but often (41 of 106 items) they had name as a substring of the destination variable (e.g. "surname"), suggesting that there may be more effective matching strategies for alternatives than simply edit distance (with the caveat earlier about "lenght" where clearly edit distance is a useful strategy).

8 DISCUSSION

The most frequently offered and most frequently executed kind of fix is that of spelling correction, but it is proportionally not selected as often as some other actions (see Table 1). Splitting the fixes by edit distance (see Table 2) shows that these fixes are selected much more often when it is only an upper/lower case difference rather than an identifier with differing characters. This suggests that case sensitivity (which is required by Java) is a common issue for many novice programmers. This may be particularly confusing for users whose native language does not have the concept of upper- and lower-case letters (such as Hebrew or Arabic).

The most frequently accepted fix was adding a class/package import, followed by adding a throws declaration or try/catch statement for an exception. We believe these are for different reasons. The necessity for imports in programs is one of book-keeping which does not require particular understanding (or lend much pedagogical benefit), has a clear outcome, and users seem happy to be assisted with this. The fixes for exception handling are taken up frequently (especially cumulatively, given they are usually offered together: 27% are taken up). This may be because the implementation is more laborious for try/catch (which involves adding several lines of code) and users want assistance – but the same is not true for adding a throws declaration, which consists of adding at most two identifiers. We believe these are taken up because users do not know the correct way to deal with exceptions and thus take the assistance because they would not know how to fix the problem manually.

In general, the fixes are taken up very quickly. The mode of the time taken between seeing and executing the fix is 1 second, and the average is under 4 seconds. This can either be taken as indication that the fixes are trivial and easily selected, or as indication that the users are doing so without thinking much about it. The general pattern of times does not differ noticeably between the fix actions, but spelling corrections and imports are selected more quickly than adding variable declarations or exception constructs. This seems reasonable: simple fixes are more quickly selected than more complex selections, lending evidence to the idea that users are thinking before clicking. The fixes are also selected with widely varying relative frequencies, from 1–17% of the time offered, which would not happen if users always executed the fix without thinking.

If users do not execute the fix, the most common behaviour is to manually execute the action of the quick fix. This provides useful confirmation that the quick fix offered is often the correct fix for the problem, but it raises the question as to why users do not use the

quick fix. It could be that the users are unclear what will happen, although this seems quite obvious (e.g., adding an import). It could be that the users do not trust the automatic fix: they worry about it messing up code style. Or perhaps they have an instinctive reaction against their code being edited for them, as if someone had taken their pen and written a sentence in their essay. It presumably is not the case that users do not understand how the fix will help given that they proceed to implement it manually. The fix text is deliberately terse to avoid unnecessary explanation, but Marwan et al. [8] found that next-step hints were taken up more if they were accompanied by an explanation. Murphy-Hill et al. [9] found that among professional developers, around 90% of refactorings (e.g., variable renames) were performed manually without using the available refactoring tools. They posit that it may be a usability issue, although quick fixes are simply clicked once, so this seems less likely. Perhaps the answer is habit: users are used to editing the code manually and they do not think the quick fix will be any faster. This has some support in the data: for correcting a name, it does seem that the manual time (5–7 seconds) is not much slower than the time taken to select the quick fix (3–4 seconds). However, seeing the fix suggestion may have enabled them to perform the manual edit faster than not seeing the suggestion at all.

An obvious potential follow-up study is to interview a sample of users to discuss their opinions about quick fixes. Are users simply ignoring them as a matter of course? Do they mistrust them or think they offer no benefit? As always with large anonymous datasets, they provide a useful starting point for a more focused follow-up study, rather than providing the complete picture by themselves.

There is also potential future work in considering if some of the fixes should be tweaked (for example, how similar names are picked for correction) or considering implementing fixes for other errors (such as missing semi-colons/brackets, or mismatched types in method calls). BlueJ deliberately has fixes that are simple and very likely to be correct (which is backed up by the data here) but this could be expanded upon in future versions.

8.1 Pedagogical considerations with quick fixes

A recurring theme in the discussion of educational development environments is the extent to which any particular tool offered helps or hinders the learning of the concepts being studied. Opponents of additional tooling usually argue that any kind of automation may enable students to create the solution without developing a full understanding of the concepts, while proponents of tools posit that additional help can prevent students getting stuck, thus increasing success and motivation, and leading to better learning experiences. This debate will only intensify with the availability of new AI-based code generation tools.

Experiences over the last decades have shown that movements to avoid or restrict available technologies in learning situations typically had little benefit beyond leaving students unfamiliar with modern software tools. No evidence has emerged that the availability of ever more sophisticated development environments has led to shallower understanding. Pedagogically, it seems clearly preferable to incorporate existing tools into a pedagogical approach than to create an artificial environment where we pretend they do not exist. Novice programmers will exit education into a world where these tools exist, so it seems sensible to prepare them to use them in a professional capacity.

9 CONCLUSION

This paper is the first detailed look at the usage of quick fixes among novice programmers. We find that the fixes are correct, where they are taken up it is within a few seconds, but the take-up rate is low, with many users instead manually implementing the quick fix instead. To expand on these points, the answers to our research questions from section 1 are as follows:

RQ1: The most frequently presented fix was to correct spelling (shown for 65% of fixable errors), followed by adding a variable declaration (32%) and adding an import (13%). Proportionally, the fixes that were most commonly executed when shown were adding an import (executed 17% of the times it is shown) followed by adding a throws declaration or try/catch statement (15% and 12%) and then spelling correction (8%). Spelling corrections were more often taken up when it was only the case (upper/lower) that needed correcting.

RQ2: Users took on average 3.8 seconds to select a quick fix, with a peak at 1 second followed by a long tail. There were statistically significant differences in the time taken between different fixes (3.5 seconds for replacing = with ==, up to 5.3 seconds for adding try/catch), suggesting that users were reading and considering the fixes rather than clicking without thinking.

RQ3: When users did not select the quick fix, the most common action was to perform a manual edit equivalent to the one offered by the quick fix. There are several possible explanations for why they do not use the quick fix: mistrust, habit, a belief that it is no faster. On the one hand, it shows that the quick fixes are offering the correct fix for an error, on the other hand, this may be a necessary but not sufficient criteria for getting users to actually use them.

RQ4: 76% of users with quick fix support available were able to fix an issue with an unimported type, compared to 52% of users without quick support available. They were also able to resolve the issue more quickly. However, fixes for an unknown variable did not change the success rate at fixing it, although it did seem to make it more likely to fix the case or add a declaration, rather than changing the name.

We believe this shows that quick fixes are worth including for users, especially in cases where the fix is clearly correct and the fix offers no particular pedagogical benefit (for example, adding a missing import). Efforts may need to be directed into understanding why users do not use them, or ways to adjust the exact implementation (for example, changing what constitutes a similar name for suggesting identifier corrections).

ACKNOWLEDGMENTS

We are grateful for the King's Undergraduate Research Fellowship (KURF) scheme which supported this project. We thank Brett Becker and Andreas Stefik for their advice on related work.

REFERENCES

[1] Umair Z. Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the Pedagogical Benefits of Adaptive Feedback for Compilation Errors by Novice Programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* (Seoul, South Korea) *(ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 139–150. https://doi.org/10.1145/3377814.3381703

[2] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[3] Neil C. C. Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 223–228. https://doi.org/10.1145/2538862.2538924

[4] Neil C. C. Brown, Pierre Weill-Tessier, Maksymilian Sekula, Alexandra-Lucia Costache, and Michael Kölling. 2022. Novice Use of the Java Programming Language. *ACM Trans. Comput. Educ.* 23, 1, Article 10 (dec 2022), 24 pages. https://doi.org/10.1145/3551393

[5] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1019–1028. https://doi.org/10.1145/1753326.1753478

[6] Robert M Kaplan, David A Chambers, and Russell E Glasgow. 2014. Big data and large sample size: a cautionary note on the potential for bias. *Clinical and translational science* 7, 4 (2014), 342–346.

[7] Mingfeng Lin, Henry C. Lucas, and Galit Shmueli. 2013. Research Commentary - Too Big to Fail: Large Samples and the p-Value Problem. *Inf. Syst. Res.* 24 (2013), 906–917.

[8] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. The Impact of Adding Textual Explanations to Next-Step Hints in a Novice Programming Environment. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 520–526. https://doi.org/10.1145/3304221.3319759

[9] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[10] Phitchaya Mangpo Phothilimthana and Sumukh Sridhara. 2017. High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students?. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 182–187. https://doi.org/10.1145/3059009.3059058