Writing Between the Lines: How Novices Construct Java Programs

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

Victoria Mac
King's College London
London, UK
victoria.mac@kcl.ac.uk

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

Novices frequently learn to program using text-based programming, in languages such as Java. Programs can be constructed in a variety of different ways. Novices may write them sequentially, one line after another, or they may use top-down design to write outlines then fill them in. They may do a lot of trial and error: writing and deleting as they receive errors, or they may use a lot of copy-and-paste. This level of detail – higher than keystroke-level editing but below plan-composition analysis – has rarely been explored in programming education research, but it can provide insight into how novices tackle program construction, and offer hints as to how they wrestle with the challenges of constructing an error-free program. In this paper we used thematic analysis to create a reliable set of tags for Java program construction, and then used them to tag over 100 programming sessions lasting over 300 hours. We empirically find that novices rarely comment, rarely plan, and frequently program by copying code they already wrote in the project and pasting it then adjusting it – which we think indicates a way to transfer the knowledge of the code they just wrote to the next code they are constructing. We offer thoughts on the viability of this work and the difficulty of tagging higher-level behaviours such as responses to errors.

CCS CONCEPTS

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

KEYWORDS

Programming traces, Blackbox, Programming education

ACM Reference Format:
Neil C. C. Brown, Victoria Mac, Pierre Weill-Tessier, and Michael Kölling. 2024.Writing Between the Lines: How Novices Construct Java Programs. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630968

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCSE 2024, March 20–23, 2024, Portland, OR, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0423-9/24/03…$15.00
https://doi.org/10.1145/3626252.3630968

1 INTRODUCTION

Learning to program involves learning to construct programs. Text-based programs can be constructed in a variety of different ways. Programmers might write them sequentially, or use top-down design to write a skeleton which is then filled in. They may comment before writing code, afterwards or never. They may make much use of copy-and-paste or none at all. This level of program construction is interesting because it may provide useful insights into novices' understanding and difficulties – as well as useful results for designers of program editors. Previous analyses have looked at areas from keystroke-level analysis [29], through compilation error counts and times-to-fix [1, 22, 27] to higher levels of abstraction such as plan-composition studies [12, 30], but investigation of line-by-line editing has seen relatively little attention.

In this paper we report on a study in which we first produced a reliable set of tags (i.e., a set of tags with good inter-rater agreement) for program editing behaviour, then used these tags to tag more than 100 programming sessions in order to see how common each behaviour is in a large dataset of novices' Java programs. We believe this will be useful for educators in understanding novices' processes, and for researchers in framing theories and studies about how novices program, as well as for designers of program editors (text or blocks) about how people construct and edit programs.

2 RELATED WORK

In order to examine and summarise the observed behaviour of programmers, especially across large datasets, many researchers have turned to developing quantitative metrics. Jadud [16] invented the error quotient for this purpose, and Shah et al. [28] constructed the Measure of Incremental Development – see Villamor [34] for a review of the area. Many more publications have reported data on error message counts, from 1976 [7] through 2015 [27] and beyond. However, this work lacks detail on what exactly novices do in response to errors. Similarly, many researchers have looked at sequences of events, such as edits and compilations, without looking at the specific content of the edits [6, 21]. There has also been work on keystroke data to look at patterns of timing and low-level behaviour [29]. Many such metrics have been used to try to predict student performance – see Hellas et al. [14] for a review.

In this paper we are interested in analysing programmer editing behaviour by looking at the content of edits. In this vein, Jadud [16] manually examined some programming traces and referred to categories created by Perkins et al. [25, 26] of "stoppers", "movers"

and "tinkerers". Hosseini et al. [15] suggested categories of Builders, Massagers, Reducers and Strugglers. Blikstein [2] suggested three categories of "copy-and-pasters", "self-sufficient" and "mixed-mode". These are slightly higher-level than we aim for in this paper; we aim to classify each edit, not the overall behaviour of the programmer. To this end, Hellas (né Vihavainen) et al. [33] examined programming traces and found that one tenth of program entry was performed by copy and paste, but they believed most of the pastes were from the student's own code. They examined very low-level entry of code, such as whether autocomplete was used or not. Gorson et al. [13] used retrospective interviews and screen recordings of programming sessions to tag certain activities which an expert system was trained to detect. The activities were relatively generic, for example "struggling with errors."

We are interested in text-based programming but there is related work on block-based editing: Dong et al. [10] classified various tinkering behaviours; Kong and Pollock [19] searched for specific two-steps sequences of edits; and Tabarsi et al. [31] created automatic detectors for certain edit behaviours within activity traces.

Reporting on an impressive case study that combined programming traces and retrospective interviews, Danielak [9] argued that "we lack... detailed accounts of how novice students design software *in the wild*..." (original emphasis). Danielak's work is in the same spirit of investigating students' programming process to examine how they actually construct program code.

In this paper we look at editing behaviours, not to classify higher-level behaviours like Perkins et al. [25], but to explore line-by-line editing, using human intelligence and judgement for the classification rather than the automated detectors of Tabarsi et al. [31]. We believe, along the lines of Danielak [9], that there is room for useful insight through human examination of editing behaviour.

3 DESIGN

For this work we used the Blackbox dataset [4]. Now over ten years old, Blackbox collects programming activity from users of BlueJ (a Java development environment aimed at novices), who are now primarily late-secondary school novices [5]. Blackbox is anonymous. Crucial for this study is that individual programming sessions (i.e. at a single machine on a single BlueJ project) can be replayed afterwards, showing the line-by-line editing behaviour and any compile-time errors that are received. This allows researchers to perform detailed *post hoc* analysis of programming sessions. The data is "natural"; although the participant will have opted in to data collection, we believe they will not have felt monitored and thus will program normally (in contrast to if a human observer was present). Moreover, many of the programming sessions are likely to have taken place outside classrooms (for coursework assignments) as well as inside them, thus providing views of programming activity that are not available by monitoring in-class.

Our research questions (RQs) were as follows:

- RQ1: How do BlueJ users construct their programs?
- RQ2: How do BlueJ users edit their programs?
- RQ3: What do BlueJ users do to their program when confronted by an error?

This study was approved according to the ethical process of King's College London.

3.1 Method

We chose to use inductive thematic analysis [3] as a basis for our research method. Thematic analysis is useful for investigating data without preconceptions in order to find common themes in the data and organise and categorise them. This is ideal for our study: we had a set of research questions to examine, but we had no pre-determined ideas of what we would find, nor any preconcieved hypotheses to test. This is an observational study, to find out how novices program by observing their programming activity, rather than by interviews or other reflective investigation.

We deliberately started with a clean slate approach rather than build on existing categories of behaviour (such as those of Perkins et al. [25]), in order to leave ourselves open to learning from the data rather than using preconceived categories. Dunne [11] provides a longer discussion of the debate over whether or not to engage with the literature before or after engaging with the data in Grounded Theory (a similar approach to thematic analysis) – in our case, one author familiar with the literature set out the research questions, while another author unfamiliar with the literature conducted the initial phase of forming categorisations. These categorisations were then refined through several phases until a high enough level of inter-rater agreement was reached, at which point the tags were used to classify a large quantity of data.

3.2 Data selection

The Blackbox dataset is too large to investigate exhaustively. Using the dataset inevitably involves some mix of filtering plus random sampling. We filtered down to only programming sessions using BlueJ 5.* (released 2021), which means the data will be homogeneous in terms of the features available to users while writing code (e.g., BlueJ changed display of error messages between version 3 and 4, and added quick fixes in BlueJ 5). We used data from the last three months of 2022; previous Blackbox research [5] has found seasonal patterns that suggest most users use BlueJ as part of academic terms (beginning in September and January), so we picked dates in term time. We took data from programming sessions that we considered to be sufficiently long for examination (many Blackbox sessions only have a handful of edit events, but this is insufficient to derive useful insights) yet not setting the threshold so high that we included outliers (who might be an especially experienced or proficient user): for the exploration stage we looked at sessions with more than 1000 Blackbox events from November 2022[1]. For the checking-agreement stage we used sessions with 500–1000 events from November 2022 (with different sessions for each round).

Once the tags had been finalised, they were used to tag many more sessions, to produce an estimate of the tags' frequency. We were concerned that if we only tagged short (300–500 Blackbox events) or long (500–1000 Blackbox events) sessions we may miss some behaviours that only occurred in the other, so we used a mix (36 long, 72 short). Similarly, we worried that if we constrained our date range too heavily, we would bias the data (given Blackbox's seasonal nature [5]), so we used sessions from October, November and December 2022 (36 each: 12 long, 24 short).

[1] We also accidentally included some data from 2013 with BlueJ 3 for the exploration phase but we do not believe this will have had a significant effect on this phase.

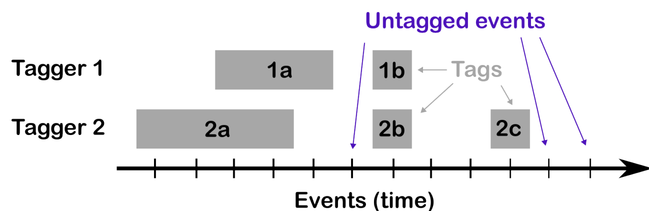

Figure 1: Example of tagging events (notches on X axis).

3.3 Checking agreement

In order to check the tags' reliability, we calculated agreement between two independent taggers using Cohen's Kappa [8]. Calculating inter-rater reliability is appropriate because we wanted the tags to be an output of the research that could be used by others in future [23]. There is a slight complication in tagging programming activity because not only do the researchers need to specify a tag, they also need to specify the start and end; there is no natural subdivision into taggable portions. This presents two issues: how to decide if two tags overlap, and what to do with portions of the sessions that no-one tagged (with our tagging scheme, there are many edit events that do not meet the criteria for any tag). For example, in Figure 1, do 1a and 2a count as agreeing, when they partially overlap? Are the untagged events counted as agreement, or ignored?

For the overlap, the popular NVivo tagging tool has a similar issue for deciding whether to count tagged sections of text transcripts as overlapping. They choose to use proportional agreement per-character, but this has been found to produce bias compared to per-sentence or per-paragraph [18]. Our tags are relatively sparse, and we chose a simple rule: if two researchers use the same tag, and it overlaps at all, we counted this as agreement (e.g. 1a-2a and 1b-2b in Figure 1). We are confident this does not introduce any false-positive agreements, as the tagged sections usually refer to a single event, or overlap substantially.

For the untagged events (some of which are labelled in Figure 1), we could not decide what was best. If we count all non-tagged sections as agreement, it inflates Kappa because we increase the number of agreements – there are many untagged portions in the data amongst the tags. If we ignore them, it deflates Kappa because we are only counting matches (1a-1b, 2a-2b) and tag-vs-no-tag disagreements (e.g. 2c), but we cast aside all the cases where the researchers agreed that no tag applied. Therefore we provide two Kappas to form a minimum-maximum estimate; one without counting untagged as agreements and one counting them as agreement.

4 RESULTS

4.1 Tag formation and validation

Thematic analysis involves a process of immersion into the data, followed by initial tag formation, then combining of tags. This was done by the second author. Then the first and second author both tagged ten sessions, of median duration 86 minutes, which produced a total of 200–250 tags from each researcher. These were then assessed against each other with Cohen's Kappa. The first round produced min-max Kappa (see subsection 3.3) of (0.33, 0.51). The researchers then examined the tag definitions together, and clarified uncertain areas, as well as splitting one tag into two. The subsequent second round of tagging ten new sessions of a similar length produced min-max Kappa of (0.40, 0.56). This mild improvement prompted a more thorough re-examination of the tags, as follows.

The researchers decided that some of the tags that were persistently low in agreement were therefore too unreliable. Although useful in theory, if they could not be tagged reliably then they are not useful. So the researchers removed tags relating to:

- Scattered coding (coding in multiple places); defining the thresholds for this proved too unclear even after two attempts.
- Rewriting (deleting code and writing something different) and Refactoring (changing the behaviour of code); these occurred rarely and also (a) small refactors proved hard to distinguish from just tinkering with the code (b) large refactors were hard to discern in the noisy process of novice programming.
- Fidgeting (changing code to no useful effect) and Polishing (small fixes to the program code); these proved hard to detect (when is it no useful effect? when is a change a polish?) and it was considered that they were of a different nature to the other tags about writing and editing code, as these tried to capture a higher-level behaviour rather than a specific act of editing.

This third round (on 10 new sessions, again of similar length) produced a Kappa min-max of (0.51, 0.65). The main reason for the low Kappa values throughout were the instances where one researcher had applied a tag, and the other researcher had not (in roughly symmetrical amounts). Where the two researchers both applied a tag, it was usually the same tag (Kappa would have been 0.82 in round 1, 0.82 in round 2, and 0.89 in round 3, for only the cases where both researchers tagged). However, 25% of each researcher's tags did not overlap a tag by the other researcher, suggesting that the problem was deciding whether an instance of behaviour qualified for a tag or not. The other reason for the lower levels of agreement was that the error behaviour tags were harder to classify. With these omitted, the Kappa min-max would have been (0.60, 0.73). We will return to this point in the discussion.

We also had a tag that did not occur during the final round of tagging. We did not find it once in our search of the full 108 sessions, so it was removed *post hoc* for being too rare:

- Pseudocode: The user writes basic pseudocode first (usually in a comment, or in code but not in Java syntax) to define the steps needed in a method or section of code.

We chose to end the tag redefinition process after the third round. We felt that any further changes would lead to minimal improvements in agreement, and further removal of tags would lead us away from our research questions. There is no definitive guideline for Kappa in all circumstances: 0.51 is usually classified as moderate agreement and 0.65 as substantial [20, 35]. Warrens [35] suggests that Kappa can be viewed as a measure of category reliability and proposes to give the reliability per-category, which we have done for tags with enough data for it to be interpretable.

The final set of 15 tags is defined in full in our codebook (see subsection 6.1 for link). A summary of the tags and their individual reliabilities in round 3 is given in Figure 2.

As explained in the full codebook, we have explicit rules for potentially-overlapping tags. For example, if code is pasted in, it is Duplication if the content of the paste already exists in the project, Moving if it was recently cut from the project, or Pasting if it does not exist in the project. If a user deletes a line, it is Error removal if the line has an error and we infer that the user was merely aiming to remove the error, but Deleting if the line has no error or deleting the line is the correct fix (e.g. if a variable declaration is duplicated, has an error about this, and is then deleted again).

4.2 Frequency of tags

The full tag frequencies, tagged by the second author, for the full set of short and long sessions (selected as described in subsection 3.2) using the final set of tags from Figure 2, are listed in Table 1. The tag counts are the number of sessions where the given tag occurred at least once. Thus one session with a hundred Deletion occurrences is counted the same as a session with one or two instances of Deletion. Where appropriate, median counts per session are also given in text here. We give a brief summary of the results here for both short and long sessions using the compact notation *Short* | *Long*.

The median duration of the sessions was 46 | 63 minutes. Recall that the differentiator of short and long sessions is the number of events; the shorter events had 63% of the events of long ones, so the duration of shorter sessions being 73% of long suggests that the shorter sessions had slightly longer pauses between events.

Naturally, many of the behaviours, especially longer behaviours such as Sequential, are more likely to occur in longer sessions: Sequential coding occurred in 29 | 61% of sessions.

Commenting was rare; only 18 | 28% of sessions had any commenting (whether pre- or post-). The median occurrences of commenting in those sessions was 1 | 2 instances.

Pasting (from outside the project) was also rare (15 | 19% of sessions, median occurrences: 1 | 2 per session), as was the Skeleton behaviour of laying out code before filling it in (7 | 11% of sessions). Duplication (copy-pasting within a project) was very common (54 | 78%, median occurrences within those: 1 | 4).

Deleting code was very common (82 | 89% of sessions), but Commenting out and Uncommenting were rare (6 | 6% and 8 | 8% respectively). Renaming variables was rare (12 | 14%).

Ignoring errors was reasonably rare (6 | 17%). It was fairly common (40 | 75%) to find and fix real errors (basic syntax and immediately fixed errors are not included in our tagging) but also common to initially be Misguided in the source of the error (14 | 33%); some sessions (0 | 14%) had both. Removing the line of code was also a reasonably common way to fix the error (26 | 44% of sessions).

5 REFLECTIONS AND DISCUSSION

This study provided many insights, both from the results themselves but also from the process and challenges of conducting the study.

5.1 The difficulty of tagging

In the course of conducting this study, we found several difficulties in tagging programming sessions.

We found that attempting to tag all activity is fruitless. When we first began we had assumed we would tag all activity, but many of the edits are either uninteresting or just uncategorisable "mush". For example, maybe the user added an extra space in a string literal or deleted (or added) a blank line. Maybe they changed an initialiser from 1 to 11 then back to 1. We could not determine a research benefit from trying to exhaustively tag such miscellaneous activity, and realised it would distract us from being able to accurately tag actual behaviours of interest.

Tagging long, higher-level behaviours was hard to do reliably. We trialled tags like refactoring, or fidgeting, and considered tags for programming by trial and error. These turned out to be hard to reach agreement on: where they started and ended, and whether they qualified. Sometimes the behaviour (e.g. refactoring) was interrupted by diversions to take care of something else (e.g. fix a separate bug), and other times it was hard to judge boundary cases: if the user rewrote the code but also made a small improvement, is that still a refactor (which we defined as behaviour-preserving)?

The low-level of agreement after the first and second rounds prompted a re-assessment of what tagging was feasible. This was not just about increasing the level of agreement for its own sake: if a tag cannot be reliably applied by multiple raters, then it is useless for building reliable knowledge. This prompted the removal of some tags. However, in retrospect, the criteria for having reliable tags did pull us more towards concrete and reliable low-level tags (like deleting, duplicating) and away from interesting but unreliable high-level tags (like being stuck, or performing refactoring).

The fact that two human raters struggled to reach agreement suggests that this kind of tagging would be difficult to automate, especially for higher-level tags. Many of the judgements are fine and contexual, with inference of what the programmer is trying to do, and has been doing. Some of the edit tags – such as Duplication – could be automatable, but we do not believe the higher-level tags (such as Misguided) are suitable for automation, neither by directly programming them nor using any kind of machine learning. However, even the low-level tags involved human judgement; for example, if the user deleted some lines but then restored them almost immediately, we did not count this as deletion. Covering all these edge cases would make automation more intricate.

5.2 Not all errors are faults

For RQ3, we examined how users responded to compiler errors. In the course of creating the tags and refining their definition we came to realise that we had implicitly sorted errors into three kinds:

- Some errors are just "slips". This consists of typos (mistyping a name), missing a semi-colon, missing brackets on a method call. These are errors, and the user was probably not aware of them until the compiler pointed them out, but they are not a failure of understanding. They are just a slip, equivalent to mis-spelling a word when writing a document. They are easily and quickly fixed, either manually or with assistance like a quick fix.
- Some errors are "premature": the user is in progress of writing a chunk of code, but an error occurs that happens because the tool attempts to compile the code while it is in progress. This is particularly acute in BlueJ 5 which auto-compiles when the user leaves the line, but it can occur in other languages or tools. Essentially, the error is something which we were confident the user already knew but was not concerned about. For example, some users write an if statement by writing { to open the body

then write the body then write }. The error that would occur in the mean time about a lack of closing curly bracket was premature, but not really a mistake. Similarly, duplicating a method ready to edit the copy may give a compile error about a duplicate method definition until the name is edited.

- The remainder are actual errors, or "faults", usually relating to types or semantics, e.g. a mismatch of types in a method call.

There are of course grey areas in identifying the exact kind of an error occurrence. The user might use a variable without declaring it, and then declare it. Is this premature (they were about to do it anyway) or a fault (they didn't realise they had to declare it until they saw the error)? It is impossible to know for sure just by looking at a programming trace.

Having formed these notional kinds, we came to realise that we did not care about tags relating to "slips", or errors that were clearly "premature" (e.g. the user wrote half a line, then did something else, then returned to complete the line). We only cared about "faults" (which may include some "premature" for reasons mentioned in the last paragraph). This formed an interesting divergence from much previous work on large datasets, which treated all compile errors equally (such as Brown et al. [4]) or which focused on syntax errors (such as Altadmri and Brown [1]) – these are usually "slips" or "premature" in our terminology and considered uninteresting.

We interpret our data on error responses cautiously, because the agreement between taggers for error responses was not high (Cohen's Kappa: 0.45). Users fairly frequently (in 26% of short sessions and 44% of long) simply deleted or commented out the offending line in order to remove the error. This would have an effect on the interpretation of some prior work on error counts (e.g. Altadmri and Brown [1]), where errors were counted as fixed as soon as they were no longer present in the (uncommented) source code. Users in general did not ignore errors (only in 6% of short sessions and 17% of long); this was less common than making misguided attempts to fix them (in 14% of short sessions and 33% of long).

5.3 Programming outwards from the familiar

Our high count of duplication matches with prior work by Vihavainen et al. [33] which found that two-thirds of pastes were from within the user's code; we found it to be higher (84%). We suggest there are three possible reasons for duplication:

Repeating yourself: Users are programming badly (the opposite of the Don't Repeat Yourself (DRY) principle) and are duplicating code that should be put into a loop or shared method.

Keystroke efficiency: Users simply want to save time and effort when writing lines of code similar to those they have already written, so they duplicate and adjust to save typing it all again.

Familiarity: Users understand the code they have already written, and they find it cognitively easier to adjust something they understand than to formulate a plan of how to write new code.

This was not distinguished in our tag definitions, but our sense was that this was not a case of "Repeating yourself"; we did not see many unrolled loops or exactly duplicated chunks of code. Users usually edited the duplicate copy to be different than the original, in ways that could not easily be avoided with helper methods.

Therefore the other two explanations remain. We believe that it is often "Familiarity" rather than "Keystroke efficiency". This is supported by our observations that when code is provided in the form of a BlueJ template, users very often delete it in its entirety and type the necessary parts again, rather than retain what they need (confirming a speculation by Brown et al. [5]). This suggests user's motivation is building code they understand, rather than a pure quest to reduce keystrokes. This accords with work such as the case study by Danielak [9] where the student reported using what they had already coded and understood as a basis for the next task (even when this is inappropriate). This is an effect which we believe deserves more attention.

5.4 Passing comment

We confirm the observation of Brown et al. [5] that comments (not counting commented-out code) are very rare; those we saw were generally either already provided in the project, or generated by BlueJ and left untouched. This will probably not be a surprise to many educators, but it was even rarer than we had anticipated; across the 108 sessions (lasting 300+ hours in total) we only saw 70 comments written, in 23 sessions. Most of these were single lines.

It could be argued that experts do not consider small programs worth commenting [24], and thus requiring students to do it is artificial. However, what works for experts does not necessarily work for beginners [17]; Vieira et al. [32] found that encouraging students to write in-code explanations of code benefitted them. Since the users we saw frequently deleted BlueJ's templated comments, it seems clear that simply putting placeholders for the users to fill in is not sufficient to encourage commenting.

5.5 Implications for program editor design

Program editor design has been a focus of research over the last decade or two, with the design of Scratch leading to interest in block-based editors and subsequently in hybrid editors. It is important for designers of these editors to know what operations programmers frequently need. For example, duplication is very common in our data, and Scratch allows duplication of code via the context menu. However, we also see a reasonable amount of moving, and it is hard in Scratch to move around individual blocks or groups of blocks within a whole. There is also no easy way to temporarily comment out a block in Scratch, although we found this is a rarer use case.

5.6 Threats to validity

The obvious threat to validity is the low level of Cohen's kappa reached during the tag validation, especially for error responses. We believe this demonstrates the difficulty of the process of tagging, especially for higher-level behaviours (such as misguided error fixing). Nevertheless, we believe our set of tags makes a reliable and useful starting point for future research, with a Kappa value indicating moderate to substantial agreement.

6 CONCLUSION

We have produced a set of tags that describe the line-by-line edit behaviour of users of BlueJ (a Java development environment aimed at novices) who are 90% student programmers with median age of 16 [5]. The tags for code construction and code editing have good reliability (Cohen's Kappa > 0.7) while error responses are less reliable (Cohen's Kappa = 0.45). We subsequently used the tags to

tag over 100 programming sessions (72 short, with median duration 46 minutes; 36 long, with median duration 63 minutes) lasting over 300 hours of wall-clock time, producing almost 1000 tag instances, in order to classify their relative frequency.

We found that the programmers frequently programmed using copy-paste within their project, duplicating existing code and editing it to suit. We hypothesise that students find this easier than writing new code, as they can build from their existing knowledge of the code. We found that commenting is very rare, with three-quarters of sessions featuring no comment writing whatsoever, and only 1 or 2 instances in each of the other quarter of sessions. Planning seems to be rare within program code: we found limited use of planning or top-down design, in comparison to just writing lines of code in a sequential fashion.

Future work may involve writing automated detectors (where possible) for these behaviours, or using the tags to classify other datasets. The tags have implications for the designs of novel block-based editors and similar, as they show which features (such as commenting out, or moving) are used frequently by novice programmers.

6.1 Replication package

We have provided our tag definitions and all our results in a public OSF repository[2]. Access to the Blackbox dataset (available on application to the admins [4]) is required for complete replication.

ACKNOWLEDGMENTS

The King's Undergraduate Research Fund helped to support this project. We are grateful to Thomas Price and Mark Guzdial for some pointers to related work.

Table 1: The percentage of Short (Sh) and Long (Lg) sessions in three different months, October, November and December (and an overall average), in which the given tag occurred at least once.

	Oct		Nov		Dec		Average	
Tag	Sh	Lg	Sh	Lg	Sh	Lg	Sh	Lg
Duplication	62	83	54	67	46	83	54	78
Pasting	21	0	8	42	17	17	15	19
Post-commenting	8	25	12	33	8	8	10	22
Pre-commenting	17	17	4	33	4	33	8	28
Sequential	21	50	29	67	38	67	29	61
Skeleton	0	0	17	17	4	17	7	11
Commenting out	12	8	12	8	0	8	8	8
Deleting	92	92	79	75	75	100	82	89
Moving	17	25	29	33	29	58	25	39
Renaming	25	8	8	17	4	17	12	14
Uncommenting	4	8	12	0	0	8	6	6
Error removal	25	42	21	33	33	58	26	44
Error seeking	38	58	38	83	46	83	40	75
Ignoring	4	25	12	8	0	17	6	17
Misguided	17	17	25	42	0	42	14	33

[2]See: https://osf.io/8fk6g/

Code construction (0.74)

Duplication: The user copies existing code they have already written within the project and pastes it elsewhere, making necessary changes to adapt to the new context. (0.78)

Pasting: The user pastes in several lines of code added at once that seems to come from an external source. (0.92)

Post-commenting: The user adds comments after writing a section of code to explain what it does or what it represents. (0.74)

Pre-commenting: The user writes comments to describe what should happen in a method or section of code before they implement its contents. (0.73)

Sequential: The user writes code in order from top to bottom of a class/method, staying focused on the same section. (0.58)

Skeleton: The user sets up a basic template of their future code, using Java syntax, with placeholders or empty space in place of concrete values.

Code editing (0.72)

Commenting out: The user comments out lines or whole sections of code.

Deleting: The user removes at least one line of program code (including comments), excluding blank lines or the line currently being worked on. (0.73)

Moving: The user moves existing code they have written into another place within the program or project. (0.59)

Renaming: The user changes the names of variables, parameters or methods.

Uncommenting: The user uncomments a line or section of code that they had previously commented out.

Error responses (0.45)

Error removal: The user removes the code with the error, either by deleting it or commenting it out. They do so with the apparent intention of just getting rid of the source of the error, rather than dealing with it. (0.35)

Error seeking: The user goes to the source of an error to fix it, making a coherent and logical attempt to solve the error. (0.54)

Ignoring: The user moves onto working on code in another part of the program or project instead of fixing any errors they encountered.

Misguided: The user keeps changing code until any errors are fixed or the code works as intended, without clear reason or approach. (0.48)

Figure 2: Short versions of the final definitions of the tags, which are grouped into three categories. Error responses can overlap with the other two behaviours, but most Code construction and Code editing behaviours should not overlap with themselves or with each other. The full definitions are found in the OSF repository (see subsection 6.1). For all categories with at least 10 instances between the two taggers, and for the three overall groups, category reliability (essentially, per-category/per-group Cohen's Kappa) is given in brackets.

REFERENCES

[1] Amjad Altadmri and Neil C. C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *SIGCSE 2015*. ACM, New York, NY, USA, 522–527. https://doi.org/10.1145/2676723.2677258

[2] Paulo Blikstein. 2011. Using Learning Analytics to Assess Students' Behavior in Open-Ended Programming Tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge* (Banff, Alberta, Canada) *(LAK '11)*. ACM, New York, NY, USA, 110–116. https://doi.org/10.1145/2090116.2090132

[3] Virginia Braun and Victoria Clarke. 2022. *Thematic Analysis: A Practical Guide.* Sage Publishing.

[4] Neil C. C. Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *SIGCSE 2014*. ACM, New York, NY, USA, 223–228. https://doi.org/10.1145/2538862.2538924

[5] 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

[6] Adam Scott Carter and Christopher David Hundhausen. 2017. Using Programming Process Data to Detect Differences in Students' Patterns of Programming. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 105–110. https://doi.org/10.1145/3017680.3017785

[7] Joan M. Chabert and T. F. Higginbotham. 1976. An Investigation of Novice Programmer Errors in IBM 370 (OS) Assembly Language. In *Proceedings of the 14th Annual Southeast Regional Conference (ACM-SE 14)*. ACM, New York, NY, USA, 319–323. https://doi.org/10.1145/503561.503628

[8] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[9] Brian Danielak. 2022. How Code Takes Shape: Studying a Student's Program Evolution. *Cognition and Instruction* 40, 2 (2022), 266–303.

[10] Yihuan Dong, Samiha Marwan, Veronica Catete, Thomas Price, and Tiffany Barnes. 2019. Defining Tinkering Behavior in Open-Ended Block-Based Programming Assignments. In *SIGCSE 2019*. ACM, New York, NY, USA, 1204–1210. https://doi.org/10.1145/3287324.3287437

[11] Ciarán Dunne. 2011. The place of the literature review in grounded theory research. *International Journal of Social Research Methodology* 14, 2 (2011), 111–124. https://doi.org/10.1080/13645579.2010.494930 arXiv:https://doi.org/10.1080/13645579.2010.494930

[12] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 211–216. https://doi.org/10.1145/2839509.2844556

[13] Jamie Gorson, Nicholas LaGrassa, Cindy Hsinyu Hu, Elise Lee, Ava Marie Robinson, and Eleanor O'Rourke. 2021. An Approach for Detecting Student Perceptions of the Programming Experience from Interaction Log Data. In *Artificial Intelligence in Education*, Ido Roll, Danielle McNamara, Sergey Sosnovsky, Rose Luckin, and Vania Dimitrova (Eds.). Springer International Publishing, Cham, 150–164.

[14] Arto Hellas, Petri Ihantola, Andrew Petersen, Vangel V. Ajanovski, Mirela Gutica, Timo Hynninen, Antti Knutas, Juho Leinonen, Chris Messom, and Soohyun Nam Liao. 2018. Predicting Academic Performance: A Systematic Literature Review. In *ITiCSE 2018*. ACM, New York, NY, USA, 175–199. https://doi.org/10.1145/3293881.3295783

[15] Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Exploring Problem Solving Paths in a Java Programming Course. In *Psychology of Programming Interest Group Workshop*. 65–76.

[16] Matthew C. Jadud. 2006. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proceedings of the Second International Workshop on Computing Education Research (ICER '06)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/1151588.1151600

[17] Slava Kalyuga. 2007. Expertise reversal effect and its implications for learner-tailored instruction. *Educational psychology review* 19, 4 (2007), 509–539.

[18] Sang-Yeon Kim, Scott S. Graham, Seokhoon Ahn, Michele K. Olson, Daniel J. Card, Molly M. Kessler, Danielle M. DeVasto, Laura R. Roberts, and Fallon A. Bubacy. 2016. Correcting Biased Cohen's Kappa in NVivo. *Communication Methods and Measures* 10, 4 (2016), 217–232. https://doi.org/10.1080/19312458.2016.1227772 arXiv:https://doi.org/10.1080/19312458.2016.1227772

[19] Minji Kong and Lori Pollock. 2020. Semi-Automatically Mining Students' Common Scratch Programming Behaviors. In *Koli Calling 2020*. ACM, New York, NY, USA, Article 7, 7 pages. https://doi.org/10.1145/3428029.3428034

[20] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[21] Juho Leinonen, Leo Leppänen, Petri Ihantola, and Arto Hellas. 2017. Comparison of Time Metrics in Programming. In *ICER 2017*. ACM, New York, NY, USA, 200–208. https://doi.org/10.1145/3105726.3106181

[22] Davin McCall and Michael Kölling. 2014. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 1–8. https://doi.org/10.1109/FIE.2014.7044420

[23] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-Rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 72 (nov 2019), 23 pages. https://doi.org/10.1145/3359174

[24] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting source code: is it worth it for small programming tasks? *Empirical Software Engineering* 24, 3 (01 Jun 2019), 1418–1457. https://doi.org/10.1007/s10664-018-9664-z

[25] D. N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research* 2, 1 (1986), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL arXiv:https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

[26] D. N. Perkins and Fay Martin. 1986. Fragile Knowledge and Neglected Strategies in Novice Programmers. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Ablex Publishing Corp., USA, 213–229.

[27] David Pritchard. 2015. Frequency Distribution of Error Messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2015)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2846680.2846681

[28] Anshul Shah, Michael Granado, Mrinal Sharma, John Driscoll, Leo Porter, William G. Griswold, and Adalbert Gerald Soosai Raj. 2023. Understanding and Measuring Incremental Development in CS1. In *SIGCSE 2023*. ACM, New York, NY, USA, 722–728. https://doi.org/10.1145/3545945.3569880

[29] Raj Shrestha, Juho Leinonen, Albina Zavgorodniaia, Arto Hellas, and John Edwards. 2022. Pausing While Programming: Insights From Keystroke Analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 187–198. https://doi.org/10.1145/3510456.3514146

[30] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (sep 1986), 850–858. https://doi.org/10.1145/6592.6594

[31] Benyamin T Tabarsi, Ally Limke, Heidi Reichert, Rachel Qualls, Thomas Price, Chris Martens, and Tiffany Barnes. 2022. How to Catch Novice Programmers' Struggles: Detecting Moments of Struggle in Open-Ended Block- Based Programming Projects using Trace Log Data. In *CSEDM 2022*. https://doi.org/10.5281/zenodo.6983260

[32] Camilo Vieira, Alejandra J. Magana, Michael L. Falk, and R. Edwin Garcia. 2017. Writing In-Code Comments to Self-Explain in Computational Science and Engineering Education. *ACM Trans. Comput. Educ.* 17, 4, Article 17 (aug 2017), 21 pages. https://doi.org/10.1145/3058751

[33] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces. In *Koli Calling 2014*. ACM, New York, NY, USA, 109–116. https://doi.org/10.1145/2674683.2674692

[34] Maureen M. Villamor. 2020. A review on process-oriented approaches for analyzing novice solutions to programming problems. *Research and Practice in Technology Enhanced Learning* 15, 1 (07 Apr 2020), 8. https://doi.org/10.1186/s41039-020-00130-y

[35] Matthijs J Warrens. 2015. Five ways to look at Cohen's kappa. *Journal of Psychology Psychotherapy* 5 (28 July 2015). https://doi.org/10.4172/2161-0487.1000197