Novice use of the Java programming language

ANONYMOUS, Anonymous

Objectives Java is a popular programming language for use in computing education, but it is difficult to get a wide picture of the issues that it presents for novices, and most studies look only at the types or frequency of errors. In this observational study we aim to learn how novices use different features of the Java language.

Participants Users of the BlueJ development environment have been invited to opt-in to anonymously record their activity data for the past eight years. This dataset is called Blackbox, which was used as the basis for this study. BlueJ users are mostly novice programmers, predominantly male, with a median age of 16. Our data subset featured approximately 225,000 participants from around the world.

Study Methods We performed a secondary data analysis that used data from the Blackbox dataset. We examined over 320,000 Java projects collected over the course of eight years, and used source code analysis to investigate the prevalence of various specifically-selected Java programming usage patterns. As this was an observational study without specific hypotheses, we did not use significance tests; instead we present the results themselves with commentary, having applied seasonal trend decomposition to the data.

Findings We found many long-term trends in the data over the course of the eight years, most of which were monotonic. There was a notable reduction in the use of the main method (common in Java but unnecessary in BlueJ), and a general reduction in the complexity of the projects. We find that there are only a small number of frequently used types: int, String, double and boolean, but also a wide range of other infrequently used types.

Conclusions We find that programming usage patterns gradually change over a long period of time (a period where the Java language was not seeing major changes), once seasonal patterns are accounted for. Any changes are likely driven by instructors and the changing demographics of programming novices. The novices use a relatively restricted subset of Java, which implies that designers of languages specifically targeted at novices can satisfy their needs with a smaller set of language constructs and features. We provide detailed recommendations for the designers of educational programming languages and supporting development tools.

CCS Concepts: • **Social and professional topics** → **Computer science education**; • **Software and its engineering** → *General programming languages*; • **General and reference** → Empirical studies.

Additional Key Words and Phrases: BlueJ, Blackbox, Novice programmers, Programming language usage

ACM Reference Format:
Anonymous. 2022. Novice use of the Java programming language. *ACM Trans. Comput. Educ.* 1, 1, Article 1 (February 2022), 31 pages. https://doi.org/10.1145/3418301

Author's Note: The Blackbox dataset has been used in several other publications, although we believe this is the first publication to use this specific subset, a thirty-two slice subset known as Blackbox Mini. The publication expands on previous work reported in a previous paper [?] (note: redacted for anonymous review) by widening the analyses and the scope of the dataset.

Author's address: Anonymous, anonymous@anonymous.com, Anonymous.

1946-6226/2022/2-ART1 $15.00
https://doi.org/10.1145/3418301

1 INTRODUCTION

BlueJ is an educational programming tool designed to help beginners learn the core concepts of object-oriented programming (OOP) [24]. BlueJ has an opinionated design: the design of the software itself and the supporting materials (particularly the popular "Objects First" textbook [6]) promote the official Java style. For example, BlueJ promotes the idea that all fields should be private and use accessors and mutators; class names should be capitalised while fields and methods are not; all classes and methods should have a header comment.

BlueJ was first released in 1999, over 20 years ago. In this time it has grown to be used by over 2 million users a year. The Blackbox data collection system [9] has been built-in to BlueJ since 2013 and allows analysis of anonymised data from all opted-in users; around half of all BlueJ users now opt in to sending their data to the Blackbox project. Data access is available to researchers, to study the activity of BlueJ users, including the source code that they write.

We pose three research questions (RQs) about Java programming style as observed in BlueJ users, given here with a justification of their importance:

- **RQ1: How much use is made of various parts of the Java language by programming novices?** This can inform discussions on language levels and language design. For example, Racket contains explicitly enforced language levels [16], as does Hedy [17], to remove more advanced concepts from the language itself for beginners. If we were to do the same for Java, what should be in the lower levels? Or, if we were to create a new language for novices, which concepts are important to include and what could potentially be left out?
- **RQ2: How effective is passively promoting a particular code style in a tool?** If everyone using BlueJ follows the intended style, the promotion would seem to be effective, but if few people follow the style, it is clear that this promotion was not effective. This will provide useful lessons for designers of other tools who may want to promote a particular code style or other practices.
- **RQ3: Do these issues of coding style and language usage change over time?** Computing is often thought of as a fast-changing field, but do the teaching practices settle into a stable pattern, or do they continue to change even for a mature language (Java is now twenty-five years old)? This will have implications for tool designers, textbook authors and course designers, as to whether and how they should update their tools/books/courses. We have access to thirty-two data slices, spaced quarterly over an eight year period from 2013 to 2021, which will let us examine possible changes over time.

This paper's contribution is threefold. First, we provide a survey of language use and style in Java "in the wild": data logged from home and classroom use worldwide, over an eight-year period of time. Second, we analyse whether BlueJ's coding style guidelines are concretely followed, and subsequently reflect on whether promoting particular code styles can be effective in a programming tool and its surrounding ecosystem. Third, we provide guidance to designers of programming languages and programming tools that are aimed at novices, based on these results for Java and BlueJ.

1.1 Related work

Significant previous work has been carried out on other aspects of novice programming behaviour in Java, for example compilation behaviour [21], compiler error messages [7], debugging [1, 8], or period of activity [10]. Errors have been a particular topic of interest, either compiler error messages or the more general topic of programming errors [20, 34]. Our interest in this paper is not in errors, but rather in use of different programming features, as well as coding styles.

There has been some previous work to look at the coding style of novices. Shneiderman [30] used multiple choice questions and fill-in-the-blank tests to evaluate Fortran programmers' behaviour, but in an artificial setting rather than actual programming. There have been experiments on how style affects comprehension [35], but these used specific examples rather than examining what students wrote themselves. There has been a large recent body of work looking at coding style and code smells in block-based languages [18, 26, 33]. Keuning et al. [23] applied a static analyser to some of the Blackbox dataset, but looked at semantic issues with coding style (e.g. missing default in switch statements) rather than the more human-oriented aspects (e.g. class naming) that we look at in this paper. Fehnker and de Man [15] used a similar static analysis approach for the Processing language. De Ruvo et al. [14] looked at Java coding style issues in submitted solutions to a small set of specific assignments. Bafatakis et al. [5] used StackOverflow as a dataset to look at style issues in Python code, although this was among experienced programmers rather than novices.

Roy Choudhury et al. [28] examined the effect of active interventions where tools provide style guidance based on the code the student wrote; in this study we instead examine the effect of a passive intervention of altering programming tool design (e.g. templates) and materials (e.g. textbooks) to encourage a particular coding style in all programs.

There have been previous studies looking at novice behaviour over multiple years. Some of them focused on particular small cohorts for a long period of time, for example Proctor and Blikstein [27] looked at 48 students for a period of 3 years. The main source of larger, online longitudinal studies has been the Scratch website. Scaffidi and Chambers [29] conducted an analysis of a set of 250 Scratch users that extended up to three years to look at their progression in use of language features. Techapalokul and Tilevich [32] looked at 100 Scratch users longitudinally although the period of time was not specified. Amanullah and Bell [4] looked at 35,000 users to track the changes in their programming usage over the course of their projects. All four of these papers were interested in tracking specific students over time, in contrast to our study which is interested in trends over time across different sets of novice programmers.

Hill and Monroy-Hernández [19] created a large dataset of Scratch projects over 5 years although they did not perform any analysis on the dataset themselves. In simultaneous work, Aivaloglou et al. [3] created a dataset of 250,000 Scratch projects but did not analyse it temporally.

There have been a few large-scale studies on particular programming constructs in block-based languages. Aivaloglou and Hermans [2] looked at their large-scale Scratch dataset for use of different constructs; subsequently, da Cruz Alves et al. [13] performed a similar analysis on AppInventor and compared their results to Aivaloglou and Hermans.

Our work in this paper is similar to the last two papers mentioned, in that it looks at language usage in a large dataset of novices. Our language is Java, which is text-based rather than block-based like Scratch and AppInventor, and we further analyse the patterns temporally over eight years to see if the patterns are stable or whether they change over time.

Some of this work was reported in a previous publication [**?**] (redacted for anonymous review) but this work extends that publication with more analyses and more data – the previous work had only two of the thirty-two slices presented here, so there is a much larger (16x) amount of data reported here with more temporal detail, as well as several new analyses.

2 METHOD

2.1 Data selection/sampling

The data used originates from the Blackbox Mini subset of the Blackbox dataset. Blackbox is a data collection project that collects anonymised interaction data of users of the BlueJ environment. The project started in 2013 and has been running (with minor interruptions) continuously since then. On the first invocation of the BlueJ system, each new user is asked whether they opt in to the anonymised collection of their data, and approximately 42% of users do so [9].

The data collected is detailed at a technical level, but limited in contextual data: The system collects interactions of users with the environment, such as source code edits, compilation events (with results and potential error messages), code executions, and use of a number of other tools. It also includes the full source code of projects, and edit sequences at line-by-line granularity.

The data, does not, however, include personal or demographic data. It also does not give information about a user's context, such as their goals or intentions. The user sample represented by the data will be strongly biased toward typical BlueJ users (novice programmers in an educational context, either at late secondary or early tertiary level), and may be biased in unknown ways by the self-selection of opting in to the data collection.

Since the Blackbox dataset itself is very large, the creators have extracted a subset known as the 'Blackbox Mini dataset' for more flexible processing and analysis. This dataset consists of 4 evenly-spaced "slices" of data per year, for the 8 years 2013–2021. The algorithm for generating each slice is as follows:

- Select a start date: 2013-06-11 was the first one (start of Blackbox), and all the others are spaced 3 months apart from this date (the latest available at the time of commencing this study was 2021-03-11).
- In each slice, only users who appear in the data for the first time after the start date of the slice are included. The first 10,657 new projects (ordered chronologically by creation date) that appear in the dataset[1] created by such users are then selected. **Thus each slice has a totally distinct set of users to the other slices.**

The full data from the 32 available Blackbox Mini slices (June 2013 to March 2021 inclusive) were used for this paper.

The Blackbox Mini database stores sequential snapshots of program code from the full Blackbox dataset in an XML format produced by the SrcML [12] tool. SrcML parses program code (in this case, Java code) and outputs an XML document representing the abstract syntax tree of the program, with tags like `<if>` and `<class>`. These source code snapshots can then be processed using any programming language/library that can parse XML. We chose to use Python for the analysis. Our analyses, which look solely at the syntax tree (as opposed to, say, the control flow graph or other transformed representations of the program), are often fairly straightforward to write when using the SrcML representation. As an example of one of the simpler analyses, Figure 1 shows the Python code for the analysis looking at the number of Javadoc comments.

2.2 Participant Characteristics

Blackbox deliberately does not collect any demographic data from its participants, in order to preserve anonymity. Therefore, we do not have exact details of the users who appear in the Blackbox Mini dataset. The best approximation of the demographics of BlueJ users originate from a 2021 survey offered to all BlueJ users, and filled in on a voluntary basis. There are a few potential caveats:

[1]The first slice was generated separately, and used the first million database events in Blackbox, which happened to correspond to 10,657 projects. The other slices were added later and were matched on the number of projects.

- The users who choose to opt in to Blackbox (roughly half) may not be the same demographic of users who opted in to the survey (roughly 3%).
- The demographic data collected on BlueJ users (September 2021 to December 2021) may not be representative of all slices during the 2021 year, or during previous years.

Notwithstanding: the survery shows that BlueJ users in late 2021 were found to have a median age of 16, with 68% male, 90% students, 70% in secondary education, and 64% were using BlueJ as part of their first programming experience.

2.3 Sample Size

Each slice included 10,657 projects from an average of 7,115 users, with a total of 341,024 projects from 227,686 users across the 32 slices. We excluded projects where none of the classes were successfully compiled within the slice: this eliminated around 7.5% of projects (with a range of 5–10% in each slice), leaving a total of 315,665 projects. One project was further excluded from one slice due to it causing issues with the parser used by the analysis code (where the code was compiled 6,826 times, causing a very large file of snapshots), leaving 315,664 projects.

The percentages shown later in each analysis graph are the percentage of projects per slice, after removing the projects with no successfully compiled classes.

2.4 Measures and Covariates

This study focuses on observing behaviours and as such we do not use statistical tests here. Instead we use summary statistics and graph trends in order to provide an overview of user behaviour within the data. There is also an issue with our relatively large sample sizes that statistical tests may turn up minor *statistically* significant results that are of little *practical* significance – a known issue with significance testing on large samples [22, 25]. As an arbitrary heuristic for practical significance, we will comment on differences that change by at least 4% (e.g. if a particular type goes from being used in 12% of projects to being used in 16% of projects) over the course of the data when looking at the smoothed trend (see subsection 2.5).

The analyses we carried out on the data are listed in subsections 2.4.1–2.4.5.

2.4.1 Javadoc comments. We counted the number of projects in each slice that included any Javadoc comments. Javadoc is a special format of multi-line Java comment, beginning with `/**`, that indicates that the comment contains documentation for the following class or method.

2.4.2 Main methods. We counted the number of projects in each slice that had a main method. A main method is the entry point for running Java programs from the command-line, but it is not required in BlueJ, which allows executing any arbitrary method from the interface.

A main method was defined as a method named "main" with a `void` return type, that was `public` and `static`, with a single argument whose type was `String[]`.

2.4.3 Naming. We counted the number of classes, variables (including fields, local variables, and parameters) and methods in each slice that were named in accordance (or not) with Java naming conventions: classes should begin a capital letter, variables and methods should not.

For classes we checked if the lead character was uppercase. For variables and methods we check that the lead character was lowercase, with the exception of `static` fields (often written as `DAYS_IN_A_WEEK`) which were skipped for this analysis.

2.4.4 Data types. We counted the frequency of use of each type in variable declarations in each slice. Variable declarations include local variables, fields, formal constructor/method parameters and for-each loop iteration variables. Lambda parameters were only counted if the type was explicitly specified (although lambda usage was very low in the data overall).

Types that were array types were counted as identical regardless of the number of dimensions. So `int[][][]` was considered the same type as `int[]` (but different to simple `int`). Generic types were counted only as their erased type, so `ArrayList<String>` was considered the same `ArrayList` type as `ArrayList<Integer>`, and neither counted as a use of `String` or `Integer`.

2.4.5 Language constructs. We counted the frequency of use of each program syntax construct (e.g. `if` statements, `while` loops) in each slice. This also included usage of different specifiers such as `public/private`, and `static`. As with all our analyses, one use in a project meant that project was counted once, regardless of how many further times it was used in that project.

2.5 Seasonal trend smoothing

After data extraction, we found that the data for most measures had a noticeable seasonal component, with periodicity of 1 year (and thus 4 slices). For this reason we decided to use STL (**S**easonal-**T**rend decomposition using **L**OESS) [11]. This procedure calculates and removes a seasonal trend, then smooths the remaining data to separate further into a smoothed trend plus a remainder term. To aid understanding, an example of the full decomposition is shown in Figure 2. For all results we will show the smoothed trend in a thick line, overlaid on the raw data in a thin line of the same colour.

2.6 Quality of Measurements

The analysis code was written by two of the authors, then independently checked by another author, to try to ensure its correctness.

3 RESULTS

The results of this paper are primarily given in graph form. To aid readability of the main text, the graphs have been grouped together on dedicated pages at the end of this paper. References to the graphs, and short text summaries of them, are given in the following sections.

3.1 Statistics and Data Analysis

3.1.1 Javadoc comments. The results for the Javadoc comments are shown in Figure 3. On average, 50% of projects have Javadoc in any of their files. The percentage of projects with Javadoc does vary over time, in the range 46–54%, with a slightly heightened level later in the date region.

3.1.2 Main methods. The results for the `main` method are shown in Figure 4. On average, 38% of projects have a `main` method in any of their files. There is a clear trend over time, with the amount of projects containing a `main` method halving between 2013 and 2021 from 52% to 25%.

3.1.3 Naming. The results for the correct naming are shown in Figure 5. A downward trend in the number of correctly named classes is apparent (i.e. named according to BlueJ's recommended code style), and a slight upward trend in the number of correctly named variables, with little change in the number of correctly named methods.

3.1.4 Data types. There are a wide variety of data types in the source code, so we have divided up the results in order to report them.

Figure 6 shows the percentage of projects that contain variables of the different primitive types, and `String`. In Java, `String` is not technically a primitive type, but it is so similar in terms of its usage frequency that we think it is best reported alongside the primitive types. There is a slight downward trend in `String`, with a larger downward trend in `double`. The alternate precision integer and floating point types (`long`, `short`, `byte`, and `float`) see very little usage.

Figure 7 shows the percentage of projects that contain variables of *arrays* of the different primitive types, and `String`. Only two types have significant usage: `String` arrays and `int` arrays, which both see a halving in usage.

Figure 8 shows the percentage of projects that contain variables of other types: those that are not primitive, not `String` and not arrays. The eight types shown are the most frequently occurring individual types when added up across all time points in the data. `Scanner` is notably the most frequently used type, and the only popular type to see an increase in usage over time. All the other types see a noticeable decrease over time, with a relatively similar pattern.

In order to display this pattern across all types (not just the eight most frequent), Figure 9 shows the previous graph with all types added up. The Y axis can be thought of as the average number of types that will appear in the project, with 100% being one type. Thus this shows that 2013 projects use an average of 3.65 types (excluding primitives, `String` and arrays), which decreases to 2.20 types in 2021.

3.1.5 Language constructs. Figure 10 shows the use of different control structures in projects. The results indicate that no control structure features in more than half the projects. There is a slow but noticeable decline in the number of if statements and while loops, and a smaller decline in for-loops.

The for-loops are further broken down into their two types in Figure 11. Java has two types of loop that use the keyword “for”. One is a “classic” semicolon loop as seen in C, C++ and many other languages in the same family, with an initialiser, condition and post-step separated by semicolons, e.g.

```
for (String line = file.read();line != null;line = file.read())
```

Since this style is very commonly used for iterating over an integer range, e.g.

```
for (int i = 0; i < arr.length; i++)
```

we recorded this use separately and report it as a sub-case. The other type of loop is a for-each loop, e.g.

```
for (String element : stringCollection)
```

As shown in Figure 11, the classic (tripartite semicolon) version is still the predominant for-loop, but around 80% of these are being used to iterate over an integer range.

Figure 12 shows the use of different exception-related items in projects. The try and catch lines being almost exactly superimposed suggests that `catch` always occurs when `try` does (in Java, this is not required – `try` can also be used only with `finally`). The `finally` construct is barely used. There is a decline in the amount of try-catch blocks and the use of the `throw` statement.

Figure 13 shows the usage of different modifiers for classes, fields and methods. For classes, there is a decline in the use of `public` but no corresponding increase in other modifiers. For fields, there is a decline in use of `static`, `final` and `public` variables. These may inter-relate as constants are often declared in Java as `public`, `static` and `final`. For methods there is a large decline in `static` methods, and a smaller decline in `public` methods.

4 DISCUSSION

4.1 Code style

BlueJ is not a tool that exists in complete isolation. There is a popular accompanying textbook [6], first published nearly twenty years ago in 2003 and now in its sixth edition. Over the years there have been an online mailing list and then community forum, as well as numerous conference presentations by the BlueJ team. All of this dissemination has advocated for a consistent style of programming with BlueJ, and we looked to investigate whether this is followed by BlueJ users.

4.1.1 Main method. The main method is the standard entry point into Java programs. Its signature is generally well-known to Java programmers:

```
public static void main(String[] args)
```

When Java is taught without an IDE, this is often taught to novices at the very beginning by telling them to memorise it, as they will need it for every program. However, since its first release, BlueJ has allowed interactive invocation of any method via its GUI, so it has never been necessary to have a main method in BlueJ. Instructors following the style encouraged in the BlueJ material tend to introduce use of the main method relatively late in the introductory sequence.

Despite this, in 2013, the main method was still being used in 52% of projects. However, the main method saw a large reduction in use over time, from 52% to 25% (Figure 4). This trend is likely to be driven by instructors and/or pedagogic material, as novices will only write a main method if someone instructs them to do so. This reduction would suggest that the instructors and/or pedagogic material authors have become more aware that the main method is unnecessary in BlueJ, even though this dataset only begins some 14 years after BlueJ's original release.

Another possible explanation of this trend is a change in the makeup of types of instructor in the user population. Two groups may be distinguished: a group of followers of the 'objects-first' style of instruction favoured and suggested by the BlueJ-related teaching material, and other instructors potentially using more traditional textbooks or online material, using BlueJ not primarily for its object interaction ability, but simply as a simpler version of a standard IDE.

The first type is likely to use the main method much less, while the latter – following widely used material – will use the main method early and often. It is possible that the second group of instructors has reduced in number in the data, as other programming languages (most notably Python) have become more popular as a Java competitor in introductory programming. Instructors not attached to an objects-first teaching style may be more likely to switch systems, while teachers favouring the emphasis on objects early may be more tempted to stay with BlueJ as an IDE. This development, if true, would shift the balance away from the use of the main method over time.

4.1.2 Commenting. Java provides a special comment format called Javadoc, which is designed to precede declarations of classes, methods and fields. In BlueJ, new classes are generated from a supplied template, which includes a Javadoc comment before the class and an example method with a Javadoc comment. All the instructional materials for BlueJ show these comments being present and explicitly recommend filling them in. Although it has remained fairly stable over time, only 51% of projects have any Javadoc in their latest version. So just over half the users are removing the Javadoc supplied with the template, but not adding it back in. Anecdotally, some users delete the entire template of the class after creating it, then type a new class from scratch – but omitting the comments.

4.1.3 Naming. There is a widespread convention in Java that class names should begin with an upper-case letter, while methods and (non-static) variables should begin with a lower-case letter. This trend is carried over into the BlueJ textbook and accompanying materials.

We see a trend in the data away from correctly named classes, but towards correctly named variables. However, if we view it in terms of upper/lower case, rather than correctness, we can consider both these trends in the same direction: away from upper-case and towards lower-case names, for both classes and fields. We speculate that this could be the influence of other programming styles (for example, although Python recommends upper-case for classes and lower-case for methods, it is often used without classes) or a general societal trend away from explicitly capitalising (e.g. because capitalisation is often added automatically during text input on mobile phones).

This raises the question whether languages or tools should require these styles rather than recommending them, or indeed whether tools should apply them as a form of auto-correction even when not typed by a user. Capitalisation may be viewed as a form of visual style similar to text colour, which is routinely applied by IDEs automatically without the need for user intervention. There is little obvious benefit in making users responsible for ensuring style consistency, and this data is evidence that they are not routinely taking on this responsibility.

4.2 Usage of the Java language

4.2.1 Types. The most commonly used number types were `int` and `double`, as shown in Figure 6. The `int` type saw stable usage while the `double` type saw a decrease from 26% to 17%. All other types saw relatively negligible use, especially by the end of the data slices, where all other numeric types were used in less than 3% of projects. We suggest that for beginners, only one integer and one floating point type is required, with all other types of differing precisions a relatively niche use case.

The use of array types is almost completely restricted to `String[]` and `int[]`, as shown in Figure 7. Both see a halving in their use over the course of the dataset. Given the reduction seen in the use of the `main` method (discussed earlier), which requires a `String[]` parameter, it seemed sensible to compare the two patterns, to try to infer how much use of the `String[]` type was driven by its use in the main method. Figure 14 shows these two patterns overlaid. The close correspondence clearly shows that the main driver of `String[]` variables is the use of the main method, and suggests that if not for this, there would only be 2-3% of projects with String arrays. Since String arrays are by far the most popular array types, we can further infer that if no projects used the main method, there would be very little usage of arrays by users, especially at the 2021 end point of the dataset.

Excluding primitive types, `String` and array types leaves all other types, which we will call class types[2]. The most popular class type is `Scanner`, which is a Java class that supports text parsing, especially from stdin (e.g. reading the next integer, reading the next word). As shown in Figure 8, this is the only popular type to see an increase in use, with all other types showing a noticeable decline. To investigate if this pattern was replicated across most class types, Figure 9 shows the sum of all class types (not just the eight most popular). This reveals that the average number of class types in each project does reduce with a similar trend across the course of the dataset from 3.65 class types to 2.20 class types, suggesting that users are favouring a smaller variety of class types on average.

Given that `Scanner` is the only popular class to buck the downward trend in the use of the number of classes, this suggests a move towards the use of text-based projects, away from the use of GUI projects (where major classes such as `JFrame` and `Color` saw a distinct reduction in use). We investigated whether the Swing GUI framework saw a reduction at the expense of the newer JavaFX GUI framework, but the small increase in JavaFX did not compensate for the much larger decline of Swing.

Although the number of types per project is small, the variety across the full set of projects is large. In each slice of 10,657 projects, a follow-up analysis showed that the median number of different types imported is 757, across a median of 250 packages. This suggests that although any individual project is using a small number of types, a wealth of types are used when we look across different projects.

[2]In Java terminology this is not technically correct but it is a reasonable definition for our context.

4.2.2 Language. The `if`-statement and `while`-loop constructs saw a noticeable decline over the course of the data, as shown in Figure 10, suggesting a trend towards smaller and simpler projects. However, no such decline is seen in `for`-loops. Separating out the two for-loop types in Java, the for-each loop is quite rare, and the semicolon variant is more popular, although most of these uses are iterating over a integer range.

The `try` construct also showed a decline (Figure 12), as did the `throw` statement, but this is primarily down to two outlier slices in the first year. The `try` construct is almost always used with `catch` and rarely with `finally`.

There was a decline in the use of `public` on classes (Figure 13) without a corresponding increase in use in other modifiers. In BlueJ, most projects contain a single package, and thus the behaviour of omitting the modifier in Java (package-private by default) is practically equivalent to writing `public`. So `public` was never really necessary in technical terms, and the data shows a trend of omitting it more as time has gone on. We suggest that is not an intentional decision on the part of the students but rather an accidental omission that is not counted as an error by the compiler. The same applies to the lack of `private` on fields; it is not a compiler error to make the fields more visible, so a student may omit an access keyword and simply not realise that it was more advisable to specify `private` access.

There seems to be a decline in the number of constants, which are usually declared as `public`, `static`, and `final`; all three modifiers have seen a sharp reduction in use with variables.

4.3 Trends over time

We have examined a variety of different aspects of users' programs, over the course of eight years. We would *a priori* expect many of the trends to be reasonably independent (e.g. capitalisation of class names, compared to say, usage of the float type). This leaves us in a position to make some inferences across multiple different patterns in the data.

All of the trends showed a noticeable seasonal "sawtooth" pattern which we adjusted for in the analysis. This shows that even with a large number of users, there are still noticeable seasonal variations during the year, so it matters which point int the year a global snapshot of programming behaviour is captured. The seasonal patterns are caused by the way that the academic year in the most frequently-occurring countries in the dataset (USA, Germany, UK) starts in September, with a break in July and August, which means that June users (at the end of the academic year) tend to be programming more advanced projects than September users (at the start of the academic year).

Once the seasonal trend was accounted for, many of the patterns did not swing back and forth over time. Many of the graphs show a relatively stable trend line wherein aspects of language use are changing gradually from a start value to an end value; not caused by a sudden change in a particular year, or trending one way for a few years and then back again. It would seem there are no fashions or fads across a large slice of novice users of mature languages, only gradual near-monotonic changes. It is surprising that even with Java and BlueJ being over ten years old at the start of the dataset, there are still such changes, suggesting that long-term usage does not quickly stabilise as we may have expected.

We also note that there are no noticeable effects in the data from the COVID-19 pandemic, which would have been visible from the March 2020 slice onwards. This pandemic caused a slight reduction in BlueJ user numbers, but it seems to have not caused any biases to appear in the data.

During the course of the eight years in the data, the Java language itself evolved. Lambdas were added in Java 8 in March 2014, and further constructs such as the `var` keyword were added in Java 10 in March 2018. We did not see much use of any of these constructs in novice programs (less than 1% of projects used lambdas), suggesting that the additions to the language are squarely aimed at professional programmers rather than novices. Language changes add a significant cost to the maintenance of programming environments for novices, because the tools generally aim to support such constructs to have full language coverage, even if the features themselves are hardly ever used by novices.

4.4 Support of Original Hypotheses

We return here to our original research questions:

- **RQ1: how much use is made of various parts of the Java language by programming novices?** In terms of language control constructs, we found relatively low usage of different constructs: only if-statements, for-loops and while-loops saw usage in over 10% of projects (individually, for each construct). In terms of access modifiers, only public and private were used, and were very aligned to their use case: around 90% of classes and methods were public while 47% of variables were private (with most of the rest of variables unspecified, taking on the default package-private behaviour).
  In terms of types, the main usage was the "quadrumvirate" of an integer type (`int`), a floating-point type (`double`), a text type (`String`) and a boolean type (`boolean`). Array types were only found to be needed by the main method and otherwise saw very little use[3]. The range of other types had a very long tail, but on average the variety of types in each individual project decreased over time.
- **RQ2: how effective is passively promoting a particular code style in a tool?** We found that BlueJ's suggested coding styles were not always strongly adhered to. Users did move away from using the main method, and they also named their variables and methods according to the lower-case convention. There was a move away from naming classes correctly, however, and only around half of projects used Javadoc comments to comment their programs.
- **RQ3: do these issues of coding style and language usage change over time?** Many of the analysis results gradually changed over time, suggesting that long-term use of a mature programming language continues to evolve – but all the changes were monotonic and often near-linear in nature over the course of the eight-year dataset, suggesting there are no fads that die out; most changes are consistent over the long-term and do not reverse.
  The meta-trend is towards simpler projects, with fewer types, and fewer control constructs. (As a follow-up analysis, Figure 15 shows the size of projects, which has reduced over time.) This may seem like a "dumbing down" but it could also reflect the changing context of novices using BlueJ for initial programming. There are two possible developments that could provide an explanation for this observation: a change in age of the programmers, or a change in other demographic characteristics.
  BlueJ started life as a system aimed at introductory university courses, but is now primarily used in secondary education. The average user has become younger as part of the general movement to teach computing in schools. This change often accompanies a movement away from computer science as an optional subject primarily taken by those with a keen interest, towards being a mandatory school subject that is taken by all students. For this cohort, a simpler introduction to the concepts may be more suitable, and this may have caused the pattern we see in the data.
  A second possible trend may be towards more casual programming, independent of age. It may be that a growing group of casual programmers has entered the landscape of learning to program, who have motivations other than becoming professional software engineers. For those types of users, code quality considerations – such as commenting and naming style – may be of less

[3]In Java, arrays are considered a low-level collection, and the use of generic types like List is generally favoured.

interest, and getting a task done as simply as possible may take precedence over exploring the entire vocabluary of the language.

4.5 Recommendations for language designers

Our dataset shows how the Java language is actually used in practice by students (primarily in secondary education, but also tertiary education). In practice, their use of the language will be heavily influenced by their instructor (and/or the curriculum). This usage pattern can be informative for those looking to design languages specifically for novices. Java itself is a professional language so it is almost inevitable that novices will use a smaller subset, but *which* subset is of interest. In this section we make recommendations for designers of programming languages for novices.

In terms of types, users primarily used four types: `int`, `double`, `boolean` and `String`. Alternative precision types saw very little use. We suggest that for novices, the main core types that are needed are one integer type, one floating-point type, boolean and a text type. Additional built-in primitive types complicate the language without adding much value for the user group in this analysis. However, library and user-defined types are important: we saw a long tail of different library types being used in each project, even though the average number of such types was only 2 per project in the most recent part of the dataset. It seems that if there are a rich set of libraries available, novices will make use of them. For language designers, this means that additional attention on the libraries may be a better time investment than more extensive systems of built-in primitive types.

In terms of language constructs, there was relatively modest usage. If statements, while loops and for loops each saw usage in over 10% of projects. But switch statements saw much less usage and are probably not worth including when if statements can fill the same need. Do-while (post-condition) loops saw very little use and could also be omitted. For-each loops were rare, but if looping over an integer range was easily done with a for-each loop as in many other languages (e.g. `for (int i : 0..9)`) then the semicolon variant could be omitted with little loss.

Try-catch statements were used, although this may be because of Java's use of checked exceptions (which must be caught or declared as thrown) in library classes, rather than a preference for using the construct itself; the throw statement saw little use itself. Many catch statements seem to catch the generic `Exception` rather than specific sub-types. Complex try statements, using the finally clause, are very rarely used. We believe novices will not use exceptions unless forced to do so by the language/libraries.

In terms of access modifiers for object-orientation, few users chose a different access modifier to the suggested default. Methods and classes could default to public, and variables default to private, without restricting the common patterns of use.

Our data shows that use of arrays in Java by novices is almost entirely driven by the use of the Java main method. Similarly, it seems that try-catch use is driven by libraries throwing exceptions rather than programmers throwing exceptions themselves. More broadly, we advise that language designers carefully consider what the core language and libraries require of the novice programmer, to avoid similar issues of adding complexity to novices' programs that is not necessary.

4.6 Recommendations for tool designers

For novice programmers, the tool they use to program is increasingly of interest. Often, like BlueJ, it is a tool for an existing professional language. Sometimes it is designed hand-in-hand with the language, such as Quorum Studio. In other cases it is indistinguishable from the language, such as with Scratch and Snap!.

Our data suggests that trying to impose coding conventions through a combination of default behaviour (e.g. BlueJ generates comments by default) and conventions (e.g. avoiding the main method, naming classes correctly) has limited effectiveness as a way to enforce user behaviour. There are two opposing design viewpoints on this: one is to say that users are making use of the flexibility of the tool, and if they want to avoid comments or name classes in lower-case then so be it. The other viewpoint is to say that if you want to promote such behaviour, you should enforce it, and design the tool to enforce the conventions. Our data suggests that putting in the conventions by default does not lead to a strong take-up by users. This design consideration may differ in educational systems for novices (where we want to teach good behaviour) compared to systems for professionals (where we might trust that they have an informed reason for disobeying convention). Overall, prolonged debate among programmers over style guides seem hardly worth the effort when a well-designed tool could easily apply them automatically (and offer individual preference setting affecting presentation where individuals care sufficiently). Style conventions primarily exist in order to increase consistency to improve readability, a goal that today seems more easily achievable through automation than by expecting consistent user behaviour.

We note that BlueJ users targeted a variety of domains, as seen in the different library types in use. We observed text-processing programs, GUI programs, web programs and many more different kinds of domains. It seems that users will make use of varied application domains when they are available within the same tool. This is in juxtaposition to the wide success of single-domain tools such as Scratch/Snap! (games/animations), App Inventor (mobile phone applications) and others.

Professional programming languages almost inexorably grow over time, adding new features to cater to new or changing demands. Java 5 added generic classes, Java 8 added lambdas, Java 10 added local variable type inference; Java 12 added switch expressions; Java 15 added records, and there is no sign of slowing down. Similar trajectories can be seen in C++, C♯, Python, Javascript and others. For those developing tools to aid use of professional languages in education, these feature additions provoke a difficult decision. Many feel drawn to support the latest language features, at the expense of initial developer time, and future development time for maintenance. Our data shows that these new features, being generally demanded by professionals, are rarely used by novice programmers. It is important to realise that there is an opportunity cost in keeping up to date: the developer time could be spend on improving the user interface or the error messages or many other ways to aid novices. We suggest that supporting new language features is often not worthwhile in education-focused tools. Good support for a limited subset of language features will be more useful for novices than mediocre support for the full language. This observation also strengthens the case for the use of dedicated pedagogical languages for programming education, the design of which could be optimised for learning characteristics instead of being driven by needs and wishes of the professional programmer community.

5 LIMITATIONS

The Blackbox Mini dataset contains data from Java programmers who use BlueJ as their development environment and choose to opt in. This creates three stages of potential bias and generalizability issues in the dataset.

First, the users are all using the Java programming language. In this study, we have deliberately focused on Java-specific items (such as Javadoc, naming conventions, and the main method). Many of these findings may transfer in spirit to other similar object-oriented languages, such as C++ and C♯. Even without such transfer, Java is also an interesting target in its own right; studies suggest that it remains one of the most popular teaching languages in multiple countries [31].

Second, the study subjects all use BlueJ. The choice of tool in an educational setting is usually determined by the educator, not the student. So our selection bias here comes from the instructors, not from the students themselves.

Finally, the users are only those BlueJ users who opt in to the Blackbox data collection. Around half of users do, but we do not know if this is a biased subset.

The lack of detailed contextual data of participants in the data collection, such as their demographic makeup, their intentions and motivations, limits the ability to generalise the findings to Java programmers in general, and imposes limits to the interpretation of the causes for the observations and trends described.

There is a potential confound with the initial 2013 data slice. This slice was collected at the launch of Blackbox. There is no way to distinguish between first-time and pre-existing BlueJ users, because they are all new to Blackbox. Therefore it is possible that users in the 2013 slice had more experience than the [new] users in the later slices, although none are guaranteed to be novices. (Much as students who come to a college introductory programming class are not guaranteed to be novices, there is no guarantee that someone loading BlueJ for the first time is or is not a novice.) This also means that their projects may have been worked on before upgrading, although the data shows that the 2013 cohort have similar sized projects to the following few years, suggesting that this may not be an issue.

6 CONCLUSIONS

The Blackbox Mini dataset gives a unique longitudinal view of general use of a text-based programming language (Java) among novice users. This allows us to examine the evolution of language use over time in a global population – not looking at individual users' learning trajectories, but rather the pattern over time across many different novice programmers.

Many of the patterns in the data show a gradual monotonic trend over time. This suggests that programming language use continues to evolve even after decades of usage, but in Java's case it is a trend towards a destination point, rather than annual swings driven by particular fads or fashions. The change may be driven by changed usage patterns of an existing user group or, as may well be the case here, a change in the demographics of the users. Ultimately, the reason does not matter; what matters more is that there are changes in requirements and expectations over time, and that tool developers would benefit from being aware of this.

The trends that are present in our case seem to suggest a move towards smaller and simpler projects, with a reduction in the variety of non-primitive types that are used, and a reduction in use of syntactic constructs, including if-statements and while-loops. This may be a reflection of the changing landscape of computing education, with more being done in schools at a younger age, more often in mandatory courses (than previously), causing instructors to aim at a simpler start for beginners.

The data supports a convincing argument that a reduction in use of the Java main method caused a direct reduction in the usage of static methods and string arrays. This suggests that the main method caused students to use features that they would otherwise not have had to deal with – the more general lesson here is that the minimal program in a language has a significant consequence on the concepts that students must deal with. There are analogues in other languages: several other programming languages also have a main method, although not always with parameters, Haskell requires use of monads in its main method, C♯ (and Java) require classes in every program, and so on.

Based on the subset of Java that novices use, we posit that novice-language designers can target a significantly simplified set of behaviours, with a few core types and a simple set of control constructs. However, users do seem to make use of a rich set of library classes and application domains, suggesting that rich libraries may be more important than a rich language. With the trend for professional programming languages to grow ever-larger in their feature sets, we suggest that educational tools for such languages are best served to target a subset of the language rather than spending large amounts of development time supporting all new features which it seems novices are unlikely to use. An alternative would be the use of a dedicated educational language that is not subject to the same market pressures.

7 ACKNOWLEDGMENTS

Removed for anonymisation.

REFERENCES

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An Analysis of Patterns of Debugging among Novice Computer Science Students. *SIGCSE Bull.* 37, 3 (June 2005), 84–88. https://doi.org/10.1145/1151954.1067472

[2] Efthimia Aivaloglou and Felienne Hermans. 2016. How Kids Code and How We Know: An Exploratory Study on the Scratch Repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. Association for Computing Machinery, New York, NY, USA, 53–61. https://doi.org/10.1145/2960310.2960325

[3] Efthimia Aivaloglou, Felienne Hermans, Jesus Moreno-Leon, and Gregorio Robles. 2017. A Dataset of Scratch Programs: Scraped, Shaped and Scored. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 511–514. https://doi.org/10.1109/MSR.2017.45

[4] Kashif Amanullah and Tim Bell. 2019. Analysis of Progression of Scratch Users based on their Use of Elementary Patterns. In *2019 14th International Conference on Computer Science Education (ICCSE)*. 573–578. https://doi.org/10.1109/ICCSE.2019.8845495

[5] Nikolaos Bafatakis, Niels Boecker, Wenjie Boon, Martin Cabello Salazar, Jens Krinke, Gazi Oznacar, and Robert White. 2019. Python Coding Style Compliance on Stack Overflow. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, 210–214. https://doi.org/10.1109/MSR.2019.00042

[6] David J. Barnes and Michael. Kölling. 2017. *Objects First with Java: A Practical Introduction* (6th ed.). Pearson/Prentice Hall. https://www.bluej.org/objects-first/

[7] 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 (ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[8] Jens Bennedsen and Carsten Schulte. 2010. BlueJ Visual Debugger for Learning the Execution of Object-Oriented Programs? *ACM Trans. Comput. Educ.* 10, 2, Article 8 (June 2010), 22 pages. https://doi.org/10.1145/1789934.1789938

[9] 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 (SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 223–228. https://doi.org/10.1145/2538862.2538924

[10] Kevin Casey and David Azcona. 2017. Utilizing student activity patterns to predict performance. *International Journal of Educational Technology in Higher Education* 14, 1 (2017), 4. https://doi.org/10.1186/s41239-017-0044-3

[11] Robert B Cleveland, William S Cleveland, Jean E McRae, and Irma Terpenning. 1990. STL: A seasonal-trend decomposition. *J. Off. Stat* 6, 1 (1990), 3–73.

[12] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. SrcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE Computer Society, USA, 516–519. https://doi.org/10.1109/ICSM.2013.85

[13] Nathalia da Cruz Alves, Christiane Gresse von Wangenheim, and Jean Carlo Rossa Hauck. 2020. Teaching Programming to Novices: A Large-scale Analysis of App Inventor Projects. In *2020 XV Conferencia Latinoamericana de Tecnologias de Aprendizaje (LACLO)*. 1–10. https://doi.org/10.1109/LACLO50806.2020.9381172

[14] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. 2018. Understanding Semantic Style by Analysing Student Code. In *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/3160489.3160500

[15] Ansgar Fehnker and Remco de Man. 2019. Detecting and Addressing Design Smells in Novice Processing Programs. In *Computer Supported Education*, Bruce M. McLaren, Rob Reilly, Susan Zvacek, and James Uhomoibhi (Eds.). Springer International Publishing, Cham, 507–531.

[16] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (feb 2018), 62–71. https://doi.org/10.1145/3127323

[17] Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER '20)*. Association for Computing Machinery, New

York, NY, USA, 259–270. https://doi.org/10.1145/3372782.3406262

[18] Felienne Hermans, Kathryn T. Stolee, and David Hoepelman. 2016. Smells in block-based programming languages. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 68–72.

[19] Benjamin Mako Hill and Andrés Monroy-Hernández. 2017. A longitudinal dataset of five years of public activity in the Scratch online community. *Scientific Data* 4, 1 (31 Jan 2017), 170002. https://doi.org/10.1038/sdata.2017.2

[20] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03)*. Association for Computing Machinery, New York, NY, USA, 153–156. https://doi.org/10.1145/611892.611956

[21] Matthew C Jadud. 2005. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education* 15, 1 (2005), 25–40. https://doi.org/10.1080/08993400500056530

[22] 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.

[23] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 110–115. https://doi.org/10.1145/3059009.3059061

[24] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ System and its Pedagogy. *Computer Science Education* 13, 4 (2003), 249–268. https://doi.org/10.1076/csed.13.4.249.17496

[25] 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.

[26] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of Programming in Scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 168–172. https://doi.org/10.1145/1999747.1999796

[27] Chris Proctor and Paulo Blikstein. 2018. How Broad Is Computational Thinking? A Longitudinal Study of Practices Shaping Learning in Computer Science. In *Rethinking Learning in the Digital Age. Making the Learning Sciences Count*, Judy Kay and Rosemary Luckin (Eds.), Vol. 3. International Society of the Learning Sciences, London, UK, 544–551.

[28] Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. 2016. Scale-Driven Automatic Hint Generation for Coding Style. In *Intelligent Tutoring Systems*, Alessandro Micarelli, John Stamper, and Kitty Panourgia (Eds.). Springer International Publishing, Cham, 122–132.

[29] Christopher Scaffidi and Christopher Chambers. 2012. Skill Progression Demonstrated by Users in the Scratch Animation Environment. *International Journal of Human–Computer Interaction* 28, 6 (2012), 383–398. https://doi.org/10.1080/10447318.2011.595621 arXiv:https://doi.org/10.1080/10447318.2011.595621

[30] Ben Shneiderman. 1976. Exploratory experiments in programmer behavior. *International Journal of Computer & Information Sciences* 5, 2 (1976), 123–143. https://doi.org/10.1007/BF00975629

[31] Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. 2018. Language Choice in Introductory Programming Courses at Australasian and UK Universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 852–857. https://doi.org/10.1145/3159450.3159547

[32] Peeratham Techapalokul and Eli Tilevich. 2017. Novice Programmers and Software Quality: Trends and Implications. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE T)*. 246–250. https://doi.org/10.1109/CSEET.2017.47

[33] Peeratham Techapalokul and Eli Tilevich. 2017. Understanding recurring quality problems and their impact on code sharing in block-based software. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 43–51.

[34] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static Analysis of Students' Java Programs. In *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30 (ACE '04)*. Australian Computer Society, Inc., AUS, 317–325.

[35] Eliane S. Wiese, Anna N. Rafferty, Daniel M. Kopta, and Jacqulyn M. Anderson. 2019. Replicating Novices' Struggles with Coding Style. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. IEEE Press, 13–18. https://doi.org/10.1109/ICPC.2019.00015

```
"""Finds the usage of javadoc comments for every successfully compiled unit in a project's file.
Two arguments are required for this script:
1) The directory in which the project directories are located
2) A CSV file to save the output to"""
import csvhandler
import datetime
import xml.etree.ElementTree as ET
import os
import sys

srcmlpath = os.path.normpath(sys.argv[1])
set_projects = set()

# Loops through projects
for (dirpath, dirnames, filenames) in os.walk(srcmlpath):
   if dirpath != srcmlpath:
         count_javadoc_files = 0

         # Loops through files in project
         for filename in filenames:
            count_javadoc = 0
            root = ET.parse(os.path.join(dirpath, filename)).getroot()

            # Only analyses project files that successfully compile
            for inner in reversed(root.findall('.//unit')):
               if inner.get("compile-success") == "true":

                  # Checks that the javadoc is not empty and that it is in the javadoc format
                  for inner_javadoc in inner.findall('.//comment'):
                     if not (inner_javadoc.text is None):
                        if inner_javadoc.get('format') == 'javadoc':
                           count_javadoc += 1
                           break
                  break

            # Counts files containing javadoc comments
            if count_javadoc != 0:
               count_javadoc_files += 1

         # Counts projects without javadoc files
         if count_javadoc_files == 0:
            set_projects.add(dirpath.split("-")[-1]) #assuming dirnames are "project-x"

# Extracts the date of the slice
year = int(srcmlpath[-7:-3])
month = int(srcmlpath[-2:])
date = str(datetime.date(year=year, month=month, day=11))

# Save data
csvhandler.CSVSaver(sys.argv[2], ["date", "projects without javadoc"], [date, str(len(set_projects))]).save_data()
```

Fig. 1. The full Python code for analysing one slice of the Blackbox Mini dataset.

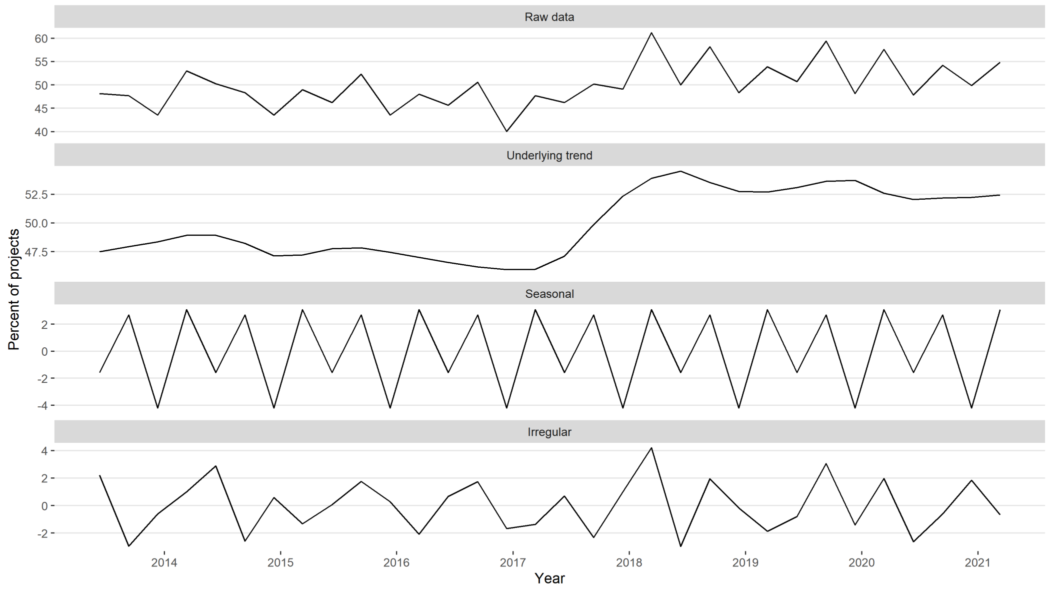

Fig. 2. An example of seasonal-trend decomposition. The raw data (top) is decomposed by first finding and subtracting the seasonal component (third item) of periodicity 4. The remaining data is then decomposed again, into a smoothed average (second item), and a remaining irregular component (fourth item). The raw data could thus be reconstituted from the other three using the equation: Raw data = Underlying trend + Seasonal + Irregular. Note that the sub-components shown here have differing Y axes to allow magnified visibility of their variation.

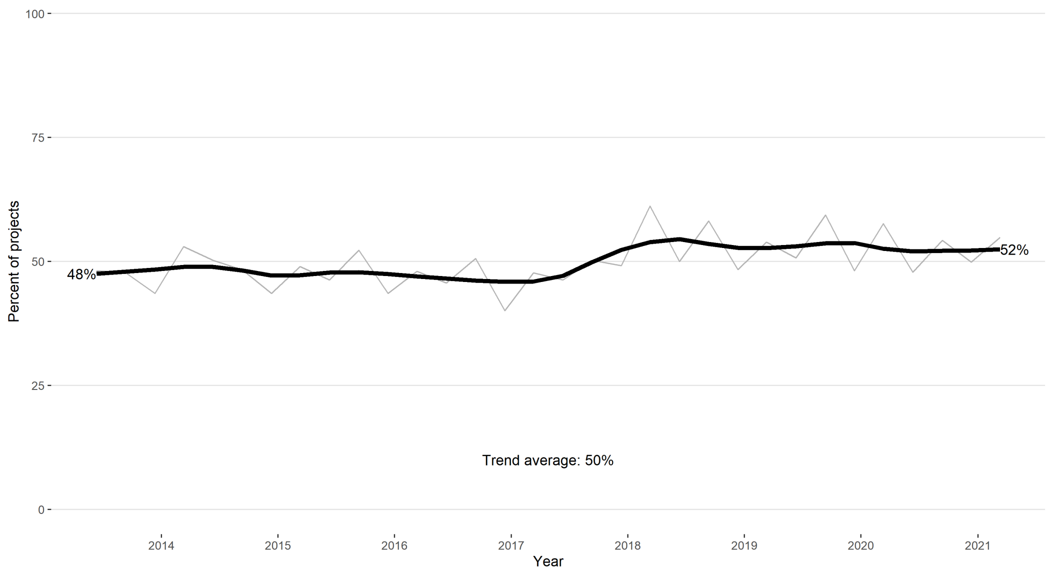

Fig. 3. The percentage of projects that contain any Javadoc (in any file).

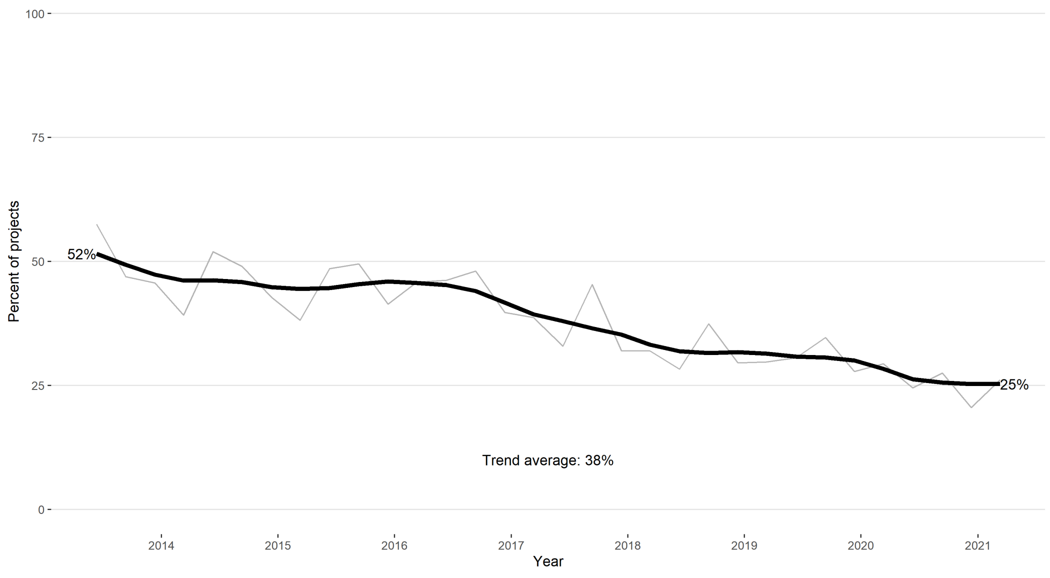

Fig. 4. The percentage of projects that contain a main method (in any file).

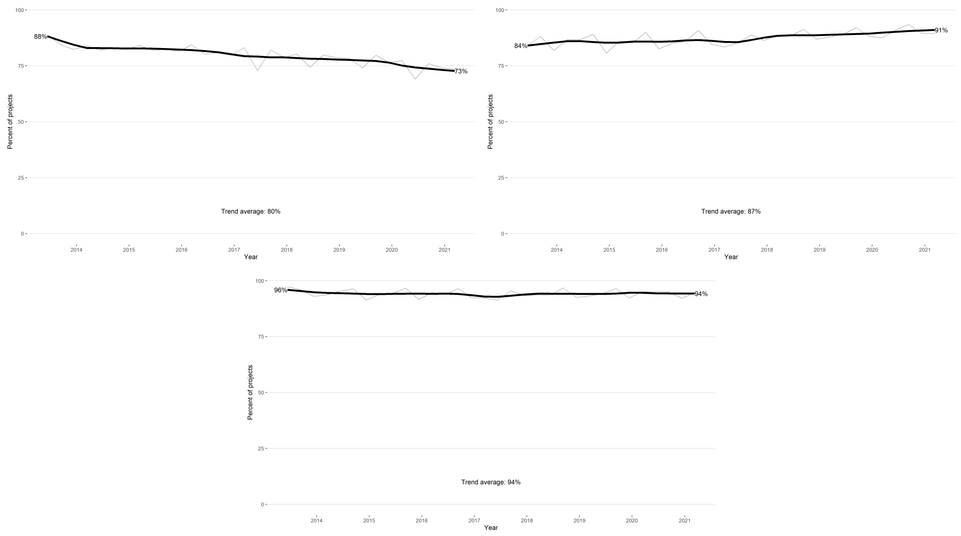

Fig. 5. The percentage of projects that correctly name all their classes (top-left), variables (top-right) and methods (bottom).

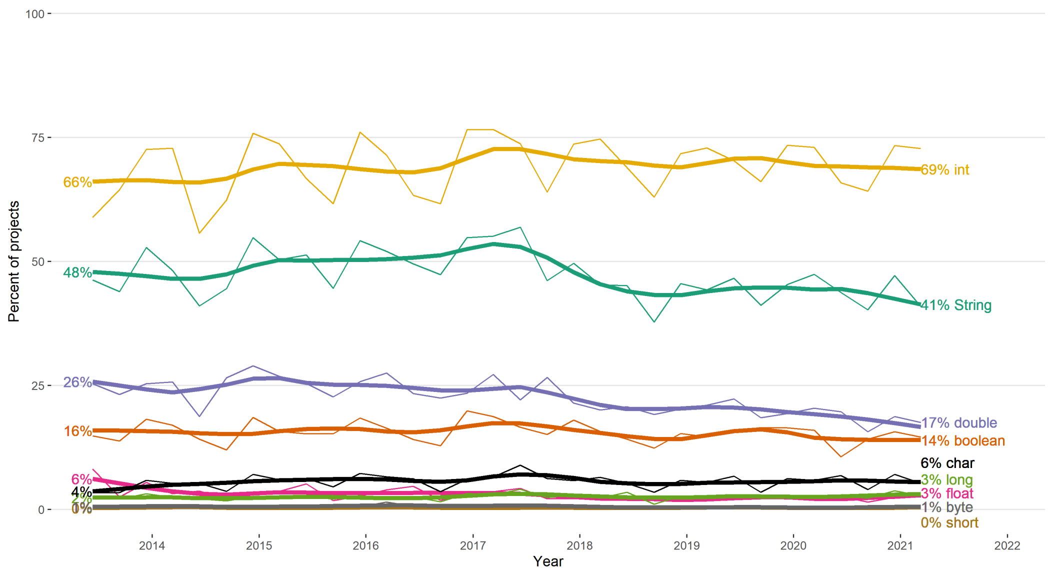

Fig. 6. The percentage of projects that have a variable declaration of the given primitive/String type.

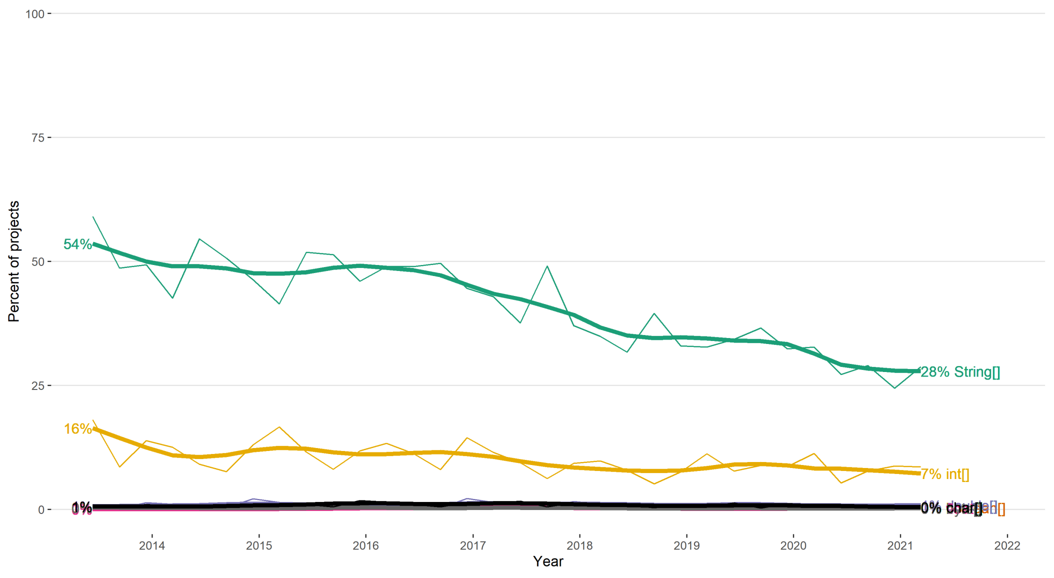

Fig. 7. The percentage of projects that have a variable declaration of the given array type.

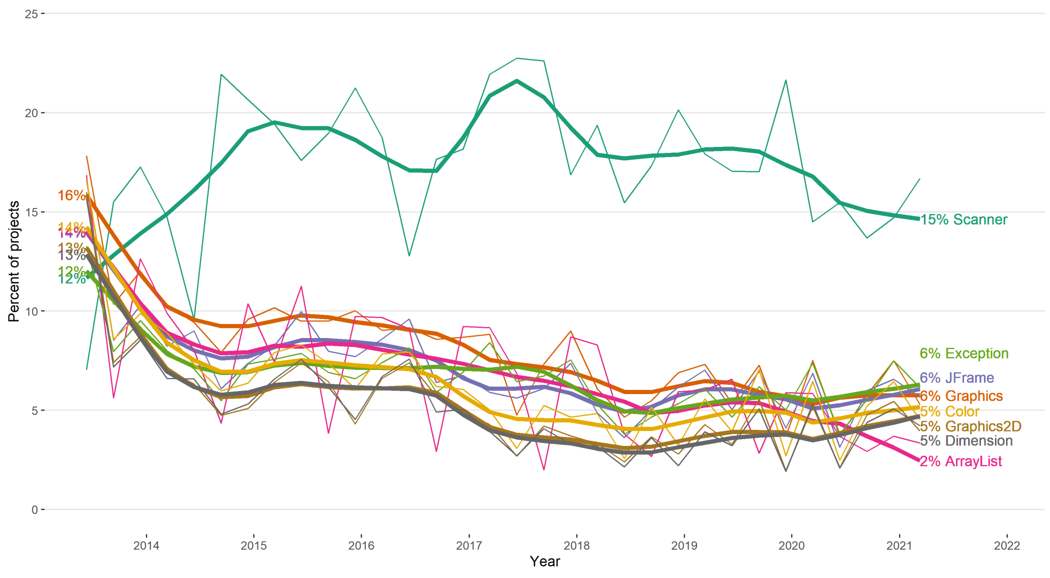

Fig. 8. The percentage of projects that have a variable declaration of the given type. Note that the Y axis does not extend to 100%.

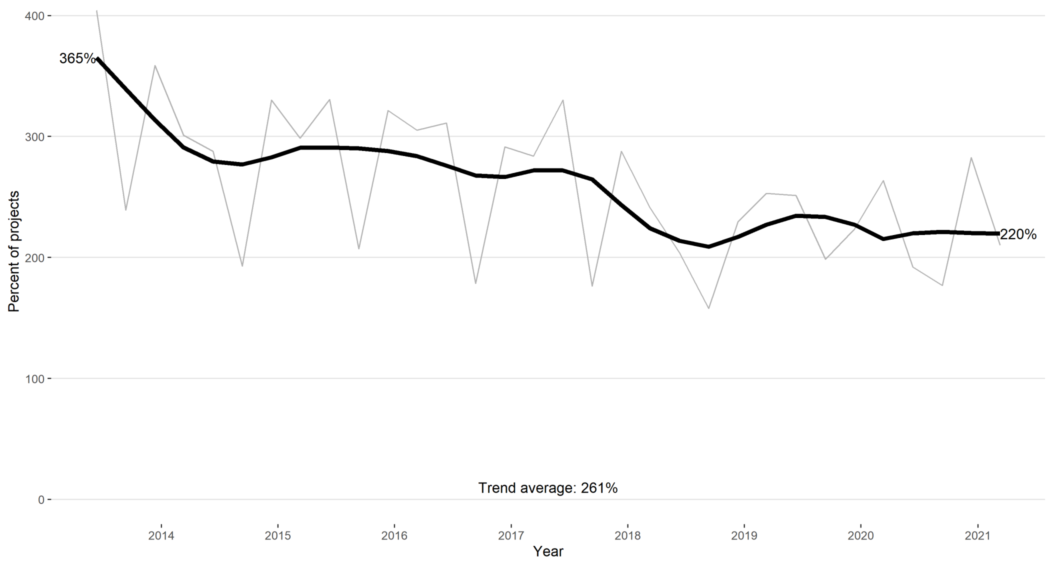

Fig. 9. The sum of the percentage of projects that have a variable declaration of the given type, across all non-primitive non-String non-array types. This can be thought of as the average number of different such types in a project, where 100% would indicate one such type.

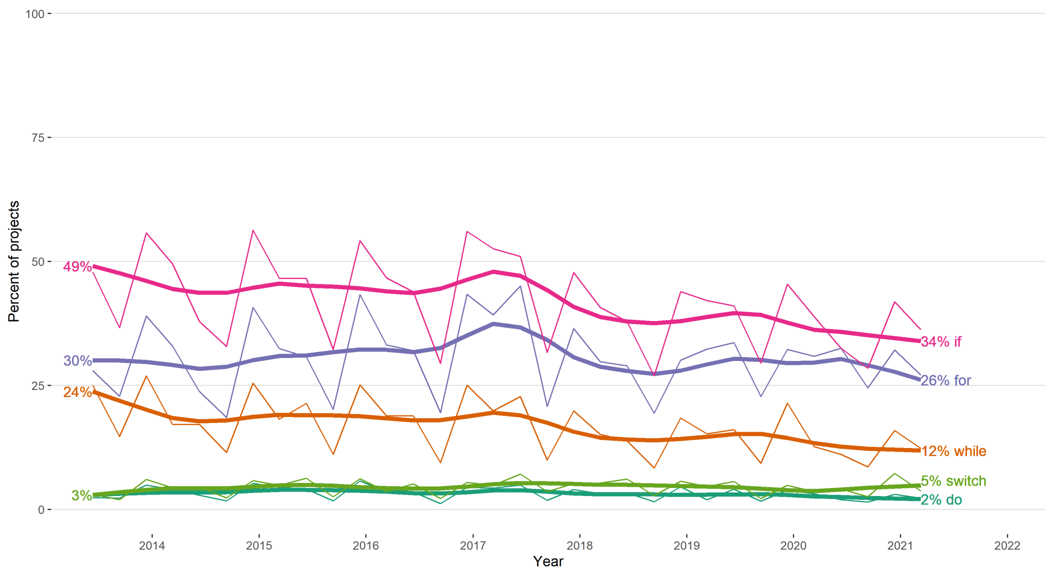

Fig. 10. The percentage of projects containing at least one instance of particular syntactic control constructs.

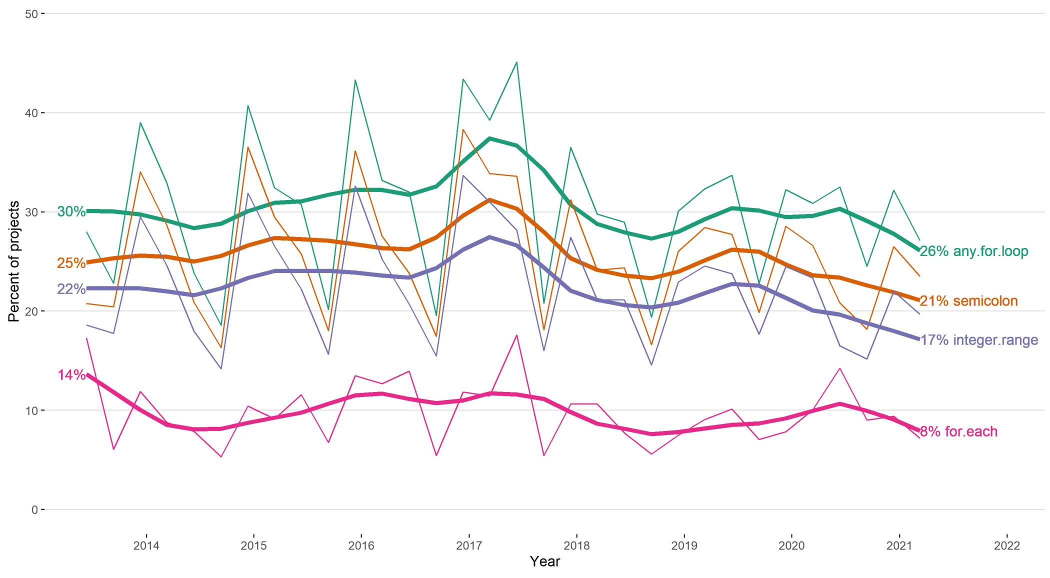

Fig. 11. The percentage of projects containing different kinds of for loops: any for loop, for loop with semicolons (C-style: `for (..;..;..)`), for loops with semicolons over an integer range (`for (int _ = _;_ < _;_++)` and similar), and for-each loops (`for (type var: collection)`). The for-each and semicolon variants will not add up to the "any for loop" because some projects will have both and are thus counted in each of the former but only once in the latter. Note that the Y axis does not extend to 100%.

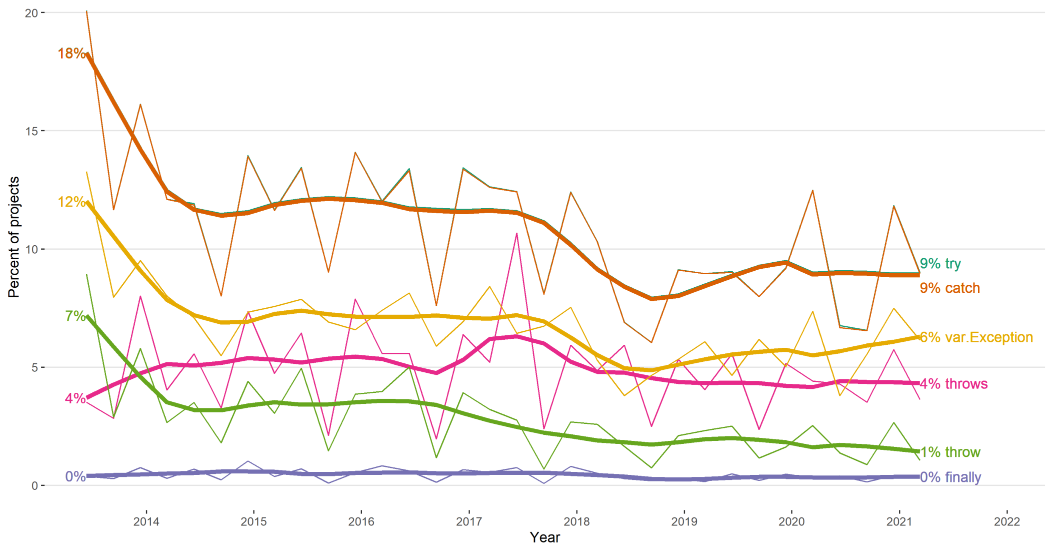

Fig. 12. The percentage of projects containing at least one instance of particular exception-related syntactic constructs. Note that the Y axis does not extend to 100%.

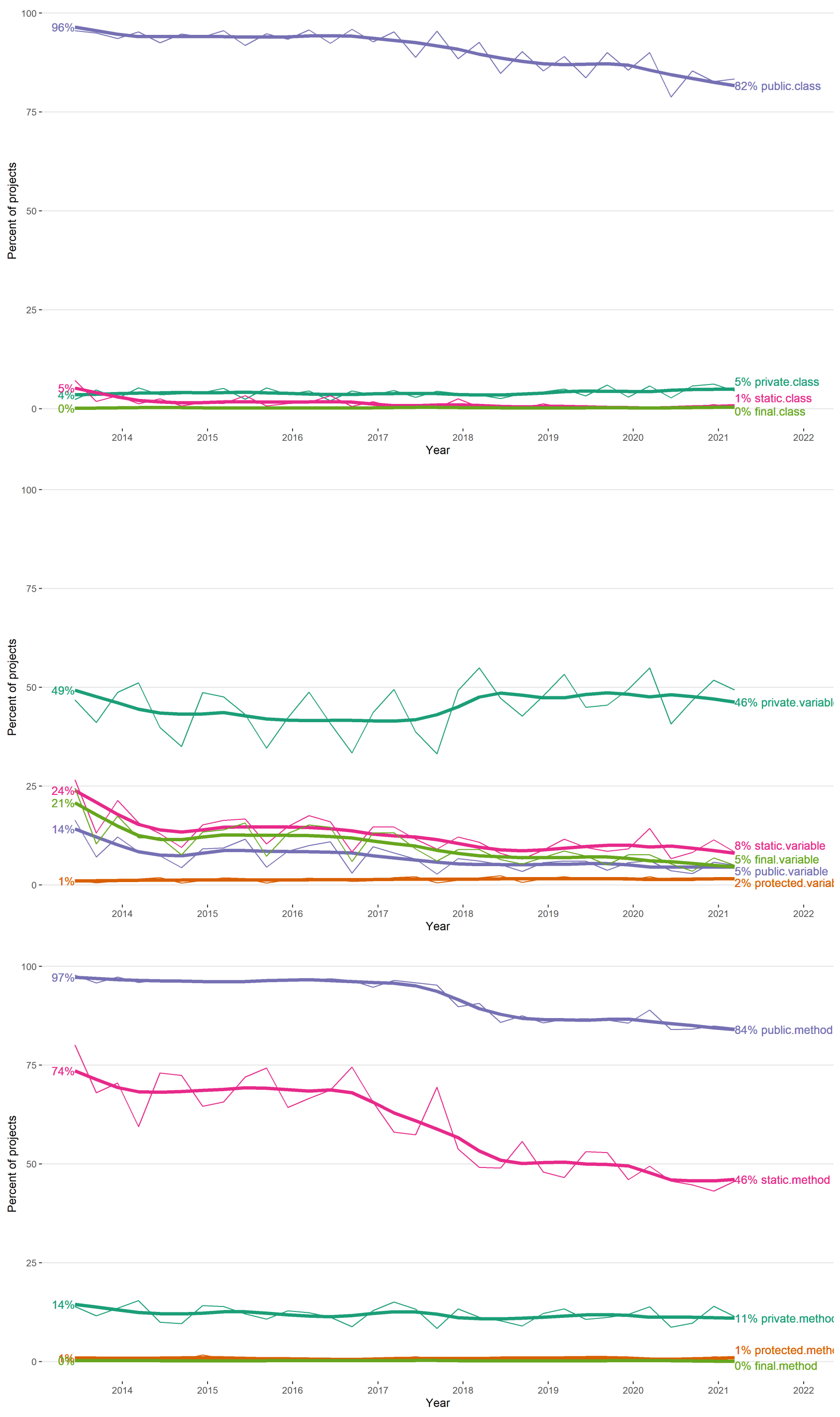

Fig. 13. The percentage of projects that use the given modifiers for their classes (top), fields (middle) and methods (bottom).

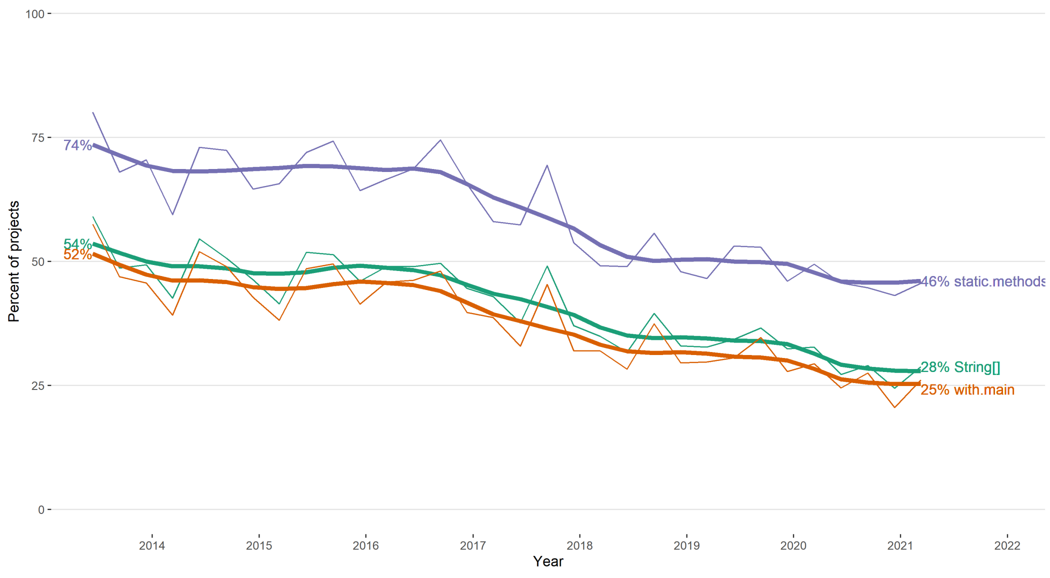

Fig. 14. The percentage of projects containing a main method with a String[] parameter, compared to the number of projects that have any variables (including parameters) of String[] type.

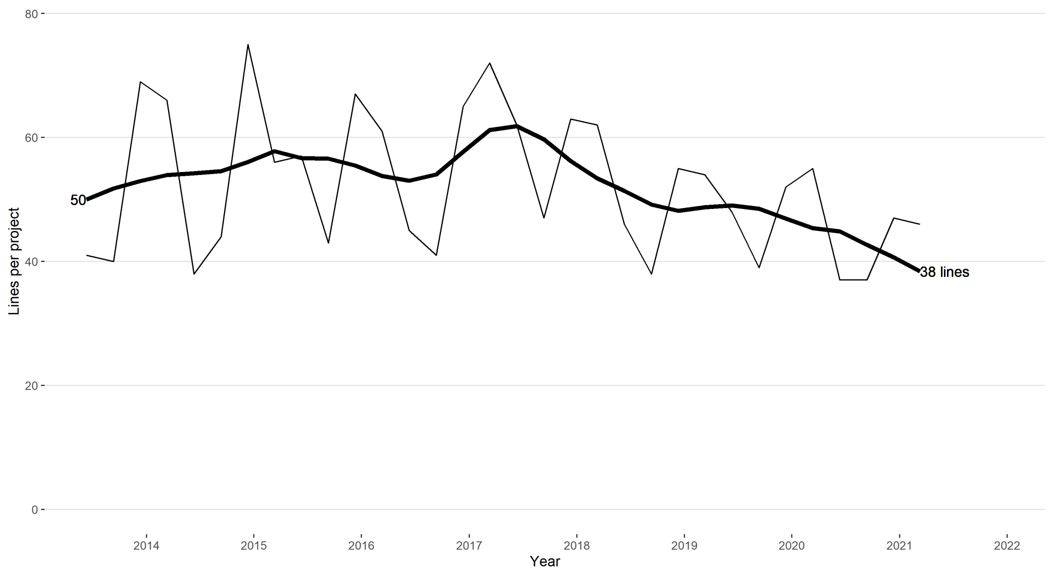

Fig. 15. The average size of each project, measured in lines of code. For each project in a slice, we calculate the total lines of code in the project (counting newlines in the source), then we take the median of all projects in a slice to get a single number of that slice.