Usage of the Java Language by Novices over Time: Implications for Tool and Language Design

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

Alexandra Lucia Costache
alexandra-lucia.costache@kcl.ac.uk
King's College London
London, United Kingdom

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

ABSTRACT

Java is a popular programming language for teaching at university level. BlueJ is a popular tool for teaching Java to beginners. We provide several analyses of Java use in BlueJ to answer three questions: what use is made of different parts of Java by beginners when learning to program; how has this pattern of use changed between 2013 and 2019 in a longstanding language such as Java; and to what extent do beginners follow the specific style that BlueJ is designed to guide them into? These analyses allow us to see what features are important in object-oriented introductory programming languages, which could inform language and tool designers – and see to what extent the design of these programming tools can have an effect on the way the language is used. We find that many beginners disobey the guidelines that BlueJ promotes, and that patterns of Java use are generally stable over time – but we do see decreased exception use and a change in target application domains away from GUI programming towards text processing. We conclude that programming languages for novices could have fewer built-in types but should retain rich libraries.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

BlueJ; Blackbox; Java

ACM Reference Format:
Pierre Weill-Tessier, Alexandra Lucia Costache, and Neil C. C. Brown. 2021. Usage of the Java Language by Novices over Time: Implications for Tool and Language Design. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432408

1 INTRODUCTION

BlueJ is an educational programming tool designed to help beginners learn the core concepts of Object-Oriented Programming (OOP) [19]. BlueJ was first released in 1999, and in the 20+ years since it has grown to be used by over 2 million users a year. The Blackbox data collection system [9] is built-in to BlueJ and allows analysis of data from all opted-in users. BlueJ has an opinionated design: the design of the software itself and the supporting materials (particularly the popular "Objects First" textbook [4]) promote a specific style of Java. For example, all fields should be private and use accessors and mutators; classes should be capitalised while fields and methods are not; all classes and methods should have a header comment.

We have three research questions (RQs) about programming style, 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 [14] 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 teaching style.
- **RQ3: do these issues of coding style and language usage change over time?** Computing is often thought of as a field of dynamic change, but do our 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 two data sets, one from 2013 and one from 2019, 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. We provide this data for 2013 and 2019, to show possible changes over 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, our analysis provides an exemplar analysis using the new "Blackbox Mini" data subset and representation, and we comment on the advantages of this data representation for program analysis.

SIGCSE '21, March 13–20, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432408

1.1 Related work

There has been much previous work on other aspects of programming novice behaviour in Java, for example compilation behaviour [17], compiler error messages [5], debugging [1, 6], or period of activity [10]. Errors have been a particular topic of interest, either compiler errors or the more general topic of programming errors [16, 24]. 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 [22] 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 [25], 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 [2, 15, 20, 23]. A previous study applied a static analyser to some of the Blackbox dataset [18], but this 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. Another study [13] used a similar approach for the Processing language. De Ruvo et al. [12] looked at Java coding style issues in submitted solutions to a small set of specific assignments. Bafatakis et al. [3] used StackOverflow as a dataset to look at style issues in Python code, although this was among experienced programmers rather than novices.

There has been previous work to examine the effect of active interventions where tools provide style guidance [21] 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.

We are not aware of any studies looking at language use over a long time period as in this study.

2 BLUEJ AND BLACKBOX

BlueJ is a Java IDE primarily designed for novice programmers [19]. The design of BlueJ focuses on conveying object-oriented concepts to the users. For example, via a representation of the class in the IDE, it is possible to interactively instantiate a class and call its methods via a contextual menu [19]. Thus BlueJ's design is based around an introduction to OOP from a direct practical interaction with objects first. In contrast, other programming approaches tend to focus on syntax first, and often leave objects until later courses because they are viewed as an advanced topic.

Several aspects of BlueJ's design specifically aim to influence programming style:

- All new classes in BlueJ are generated from a fixed template which has a Javadoc comment for the class and method. This encourages users to retain such comments, and aims to start a habit of always providing such a comment.
- The template always has access specifiers (public, private, etc) on classes, fields and methods. In Java it is usually possible in single-package programs to omit all such specifiers (the default being package-wide visibility) and still have a program compile, but this template hints that the access specifiers should be given.
- There is no need for a main method (the infamous **public static void** main(String[] args)) in BlueJ. Code can be executed by constructing an object or invoking a method in the GUI. Not only does this mean that students can avoid the complicated main method syntax at first, but it also moves away from the command-line style of programming that the main method (with its command-line arguments) implies.

BlueJ's ecosystem also comprises several resources for educators in accordance with the aforementioned principles. The BlueJ textbook [4] is co-written by BlueJ's creator and proposes a pedagogical approach of OOP by providing code project examples and exploring the functionalities of BlueJ. The book's authors present OOP through projects and software engineering concepts (e.g. designing classes, handling errors) rather than simply referencing Java language constructs, and they provide a specific code style that is adhered to throughout the book and explicitly given in the appendix. The Blueroom is an official BlueJ teacher's community website where members can share teaching resources; the most popular resources are those written by BlueJ's creators, which again espouse the same coding style.

2.1 Blackbox

From BlueJ 3.1.0, users can opt-in to take part in an anonymised data collection project named Blackbox [9]. The data is sent from the users' machines to a central server where it is stored in an SQL database. Blackbox is available to other researchers and has been used for several studies [5]. We presented a summary of its usages up until 2018 in a retrospective paper [7].

Inspired by feedback from researchers that the data schema is unwieldy, we introduced Blackbox Mini at SIGCSE 2020 [8]. Blackbox Mini contains a subset of the Blackbox data-set. The source files are stored in srcML [11], an XML-based markup language created by Collard and Maletic to represent the source code of several programming languages. Blackbox Mini data can be analysed using any XML processing techniques, for example using XPath queries. Blackbox Mini now comprises two slices: one that covers the first million events of the BlackBox data-set (in 2013), and another covering a million events of 2019.

3 ANALYSES AND RESULTS

To answer our research questions, we performed multiple different code analyses. Our RQ1, concerning the use of different Java language features, is investigated by looking at:

- use of different Java data types in the source code, and
- use of different Java keywords and constructs in the source code.

Our RQ2, concerning promotion of the BlueJ preferred Java coding style, is investigated by looking at:

- use of Javadoc comments,
- adherence to naming conventions (class names should start with a capitalised letter, method and variable names should not),
- the use of the standard **public static void** main (String[] args) method in the project (not needed in BlueJ),

Our RQ3, concerning change over time, is investigated by performing all of the analyses on both the 2013 and 2019 data set, and then comparing the results.

```
countJavadocFiles = 0
for filename in filenames:
  countJavadoc = 0
  root = ET.parse(os.path.join(dirpath, filename)).getroot()
  for inner in reversed(root.findall('.//unit')):
    if inner.get("compile-success") == "true":
      for innerJavadoc in inner.findall('.//comment'):
        if not (innerJavadoc.text is None):
          if innerJavadoc.get('format') == 'javadoc':
            countJavadoc += 1
      break
  if countJavadoc != 0:
    countJavadocFiles += 1
if countJavadocFiles != 0:
  print "This project has javadoc."
else:
  print "This project does not have javadoc."
```

Figure 1: Example snippet of analysis using Python and XPath on srcML files to determine whether any files contain Javadoc comments. As this example shows, such analyses only require a few lines of code and simple XPath queries (for example, ".//comment" finds all comments in the source within the current node).

3.1 Data and Method

The data used is the Blackbox Mini subset of the Blackbox database. There are two data sets: one from the start of Blackbox in 2013, and a matched data set in 2019. The 2013 data set covers the first million events of Blackbox, from June 2013, and has 10,657 projects containing 24,217 files, belonging to 10,455 users. The 2019 data set is matched on the time of year and number of projects, and only contains projects from new users. It has 10,657 projects containing 31,285 files from 10,576 users. Each Blackbox project belongs to exactly one user, so the number of projects-per-user is 1.02 and 1.01 for the two data sets. Since this ratio is so close to 1, we report only proportion of projects here, but this can also be understood as the proportion of users.

The following results have been retrieved using XPath expressions and Python to process the srcML files in the BlackBox Mini data set. To give an idea of the kind of analysis that is used with the srcML format, the source code of an example analysis is given in Figure 1. The analyses examine only the last successfully compiled version of each source file. If no such version is found, the project would have beeen dropped from the statistics; however, all files have at least one successful compilation in both the 2013 or 2019 data sets.

3.2 Javadoc Comments

BlueJ's guidelines [19] give prominence to the use of comments in code. If a user creates a new class in BlueJ, the default template contains a Javadoc[1] comment for the class and the method. The only way to have a class without these is to later remove them, or import an existing source file without them into a BlueJ project.

Our analysis found that 48.6% (5,179) of 2013 projects did not have Javadoc, and neither did 46.1% (4,918) of 2019 projects. This indicates a stable pattern over time of half the projects lacking Javadoc. In light of the aforementioned templating, this means users are actively removing the Javadoc and not putting it back. We do see instances in the data of users deleting the entire templated content and typing in a new class from scratch, which indicates that templates in programming IDEs are actively "resisted" by users, and may not be very effective – and that users are relatively unswayed by widespread Java guidelines telling them to provide such comments. In case the issue was that users were using comments without the Javadoc syntax, a brief follow-up analysis looking for class header non-Javadoc comments was performed, but did not show any difference to this result.

3.3 Main method

When running Java from the command line, it is necessary to have a main method with the signature **public static void** main(String [] args). This was often held up as an example of unnecessary verbosity and confusion for Java beginners. BlueJ does not require this method because any method can be directly invoked, and it is only introduced later in the Objects First textbook [19]. However, students using other material may be told to include it, and if the students want to export the program to run outside BlueJ, it would also be necessary.

We found that in 2013, 53.7% (5,721) of projects had a class with a main method, and in 2019 the figure was 28.8% (3,065). This suggests that users may now be relying less on teaching materials that require adding a main method.

3.4 Wrong naming conventions

The Java naming convention, which is mirrored in the Objects First textbook and all BlueJ examples, is that class names should start with a capital letter, while fields and methods should not.

For the 2013 data set, 10.5% (1,115) of all projects did not have proper class names; in 2019 this was 24.2% (2,583). For the method names, in 2013 there were 2.6% (276) of all projects without proper method names and in 2019 it was 5.4% (574). For field names, in 2013 there were 26.1% (2,783) of all projects without proper field names, and in 2019 this was 12.4% (1,324). So although the misnaming of classes and methods doubled, the number of misnamed fields halved. One way to look at this is rather than a trend towards incorrect capitalisation for each, instead it is a trend in the classes and fields to always use lower-case names (which are considered wrong for Java classes, but correct for Java fields). This may come from other programming languages where lower-case is common (e.g. Python) or perhaps a trend in keyboard typing to not use capital letters.

3.5 Data type usage

Java variables (including local variables, fields, parameters, etc) must always be declared with a specific type[2]. We can look at

[1] Javadoc is a specific comment format beginning /** that indicates a documentation comment.

[2] Since Java 11 – supported in BlueJ since 2019 – this is not technically true. The new var keyword allows local type inference. We only found 11 projects that used it in 2019, which is not large enough to affect our results, so we will ignore it in this analysis.

Data Type	2013 Projects	2019 Projects
int	5,881 (55.2%)	7,011 (65.8%)
String	4,622 (43.4%)	4,650 (43.6%)
double	2,532 (23.8%)	2,222 (20.9%)
boolean	1,484 (13.9%)	1,560 (14.6%)
float	813 (07.6%)	293 (02.7%)
char	336 (03.2%)	672 (06.3%)
long	229 (02.1%)	274 (02.6%)
byte	43 (00.4%)	60 (00.6%)
short	30 (00.3%)	62 (00.6%)

Table 1: Frequency of all primitive data types and String, in descending order of their combined total. Each count is the number of projects that contain at least one variable declared with that type.

Construct	2013 Projects	2019 Projects
if	4,782 (44.9%)	4,090 (38.4%)
for-loop	2,797 (26.2%)	3,354 (31.5%)
while loop	2,500 (23.5%)	1,606 (15.1%)
try	2007 (18.8%)	902 (08.5%)
throw statement	894 (08.4%)	250 (02.3%)
throws declaration	352 (03.3%)	546 (05.1%)
switch	305 (02.8%)	564 (05.3%)
do-while loop	237 (02.2%)	406 (03.8%)

Table 2: Frequency of selected program constructs, in descending order of their combined total. Each count is the number of projects that contain at least one of that construct. Note that the for-loop entry includes both types of Java for-loop: C-style and for-each.

the types used for variable declarations to get an impression of which types are used by programmers. Note that because we look at declarations, this code will not count as having an int:

```
System.out.println(7);
```

because there are no int variables, but this one will:

```
int x = 7;
System.out.println(x);
```

Although the behaviour of the code is equivalent, they are different from a student's perspective, because they have had to explicitly think about the type and declare the variable in the second case. So we feel it is justified to count the second case as using an int but not the first. We count the number of projects using the type rather than the instances of the type, because otherwise a project with many uses of a particular type could bias the statistics. As mentioned earlier, the number of projects using the type is also an effective proxy for the number of users using a type. The frequencies for primitive types and String[3] are shown in Table 1.

To give a comparison, the other most frequent data type names in 2013 were Graphics (1,782 projects), ArrayList (1,684) Color (1,664) and JFrame (1,660); in 2019 they were Scanner (1,700 projects), ArrayList (653), Graphics (535) and JFrame (501). This list has a long tail, with 2,409 different data type names in 2013 and 3,101 in 2019. Many refer to types declared within the current project and thus have frequency one. Primitive types (excluding String) accounted for 50.2% of all variables in 2013, and 51.9% in 2019.

The number of projects that contained a variable declared with a standard collection class (which are supplied in the Java standard libraries) type were as follows: List (265 projects in 2013, 353 in 2019), ArrayList (1,684 in 2013, 654 in 2019), Map (56 in 2013, 108 in 2019) and HashMap (172 in 2013, 206 in 2019). This suggests that users usually use the concrete type (ArrayList, HashMap) for a variable's type, rather than the canonical interface (List, Map). Iterator was used in 209 projects in 2013 and 192 in 2019.

We also looked specifically at type use in catch statements, which changed between 2013 and 2019. In 2013 the most frequently used types in catch blocks in projects were IOException (1504 projects), Exception (1326 projects), InterruptedException (500), NumberFormatException (79). In 2019 the most frequent were Exception (464 projects), IOException (310), InterruptedException (26), NullPointerException (25). We discuss this later in the paper.

These results can provide some insight into the changes in GUI programming over time; in 2013, Java's primary toolkit was Swing, while in 2019 in theory JavaFX is the best option (in this time, BlueJ switched from Swing to JavaFX). Looking at the window classes as a guide, Swing's window class JFrame went from being used in 1,660 projects in 2013 to 501 projects in 2019, while JavaFX's window class Stage went from 0 in 2013 to 118 projects in 2019. This suggests that JavaFX in general has not yet caught on, and also hints that GUI programming may have declined as a use case in BlueJ in that time. For comparison, Scanner (a class commonly used for text parsing) went from 704 in 2013 to 1,700 in 2019.

3.6 Java constructs and modifiers

To get an overview of the use of different aspects of the Java language, we looked at the use of Java keywords in particular program constructs (e.g. while-loops) and modifiers (e.g. static, or private).

A ranking of different program constructs is provided in Table 2. From 2013 to 2019 we see a shift away from while loops towards for-loops and do-while loops. The switch (multiway comparison statement, often called select or case in other languages) statement also sees an increase, but remains infrequently used.

We also see a halving of the use of the try statements. We have also included some relevant further statistics: the use of the throw statement (which throws an exception) has decreased markedly, but the use of the throws declaration (which indicates that a method may throw an uncaught exception) has increased. This suggests a decreased use of exceptions, and a decrease in catching the exceptions that do occur.

Several Java keywords are very infrequently used by beginners: strictfp, native, transient, assert, volatile all have under 20 projects using them in each year, and synchronized is similarly rare.

Many specifiers in Java (e.g. private) can be applied to multiple different constructs (e.g. classes, variables), so we separate these out: we provide counts of use for class specifiers, method specifiers and variable specifiers in Table 3. We note some patterns here.

[3]String is not a primitive type in Java, but it is so commonly used that we feel it deserves treating like one here.

	Classes		Methods		Variables	
Specifier	2013 Projects	2019 Projects	2013 Projects	2019 Projects	2013 Projects	2019 Projects
public	9,548 (89.6%)	8,339 (78.2%)	9,766 (91.6%)	8,650 (81.2%)	1,641 (15.4%)	597 (05.6%)
protected	3 (00.0%)	0 (00.0%)	95 (00.9%)	132 (01.2%)	124 (01.2%)	132 (01.2%)
private	235 (02.2%)	323 (03.0%)	1,392 (13.1%)	1,065 (10.0%)	4,675 (43.9%)	4,480 (42.0%)
static	716 (06.7%)	84 (00.8%)	8,001 (75.1%)	5,299 (49.7%)	2,661 (25.0%)	949 (09.0%)

Table 3: Frequency of specifiers for classes, methods and variables. Each count is the number of projects that contain at least one of that specifier applied to a class, method or variable respectively.

The amount of static methods decreased from 75.1% to 49.7% between 2013 and 2019. It should be remembered that in 2013, 53.7% of projects had a class with a static main method, and in 2019 the figure was 28.8%. So most of this drop in static methods is accounted for by the reduction in main methods. The drop in static classes and fields may be driven by this, too – if a user's primary code is in a static main method, they are likely to make their variables static to be able to access them, but this is not necessary if their code is in an instance method.

Protected is very rarely used; public is the most common specifier. In BlueJ most projects are a single package, so public is not necessary (you can omit the specifier to get package-wide access in Java) on classes for the code to compile. Therefore users are retaining the public from the class template or re-entering it. This is in contrast to the earlier Javadoc finding where users were removing things provided by the template. The use of public fields (generally considered bad practice, and never used in the BlueJ coding style) declined noticeably between 2013 and 2019, from 1,641 down to 597. Many of these public fields are likely to be static constants.

3.7 Lambda Expressions

We briefly examined the use of lambda expressions. Lambda expressions were introduced in 2013, so they were only present in the 2019 data set. Only 0.27% of projects from the 2019 data set had lambda expressions, suggesting they are rarely used by beginners.

4 DISCUSSION

In this section we relate our results back to our research questions.

4.1 RQ1: how much use do novices make of various parts of the Java language?

It is unsurprising that some obscure keywords (e.g. native methods) are unused by beginners, as they are intended for expert use. But there are other intermediate concepts that go almost unused, including synchronized access and the new lambda expressions. We would expect the use of lambda expressions may still grow in future, but six years after their release, only 0.3% of projects are using them, suggesting that most Java novices and educators view lambdas as too advanced for beginners. Functional programming educators will no doubt disagree.

The post-condition do-while loop is less popular than the pre-condition while loop, but the gap is narrowing. If-statements are the most popular construct, but interestingly only around 41.6% of projects feature them. We had expected this number to have been much higher; this perhaps indicates that selection is not as important for initial programming as we had expected.

The most commonly used data types for variables are int and String, followed by double and boolean. There is a fairly convincing argument for dropping most of the rest of Java's built-in types for novices. The other integral types (byte, short, long) are rarely used, and float has declined in usage. Obviously this is not feasible for the existing Java language, but designers of new languages for beginners would be justified in offering one integral type, one floating point type, a boolean and a string type.

Container classes were popular, but users preferred to declare variables with the concrete type (ArrayList, HashMap) rather than the abstract canonical interface (List, Map) as some educators recommend, perhaps due to a lack of understanding of the principles of inheritance. Usage of iterators was constant between 2013 and 2019, suggesting that the slow move from iterators to for-each loops started by Java 5 in 2004 has now completed.

4.2 RQ2: how effective is passively promoting a particular code style in a tool?

BlueJ contains various measures, especially its class templates, to promote good practice, and this is consistent across supporting materials. This seems to not be fully effective in imposing a code style. From a user's perspective, this is fine: BlueJ promotes one style but will accept all valid Java. From a designer's perspective, it is disappointing that such passive measures do not seem to have a strong impact. Despite BlueJ not needing a main method, over half of the users in 2013 still supplied one – although this decreased to just over a quarter in 2019. This strong shift seems surprising when BlueJ and Java were both already over ten years old.

Educators will probably be unsurprised that students do not comment well, but our data shows that even when given the comments, students will remove them rather than leave them in the source code. The treatment of access permissions was much more in line with commonly accepted code practice: private fields were more commonly used than public fields, and this improved over time.

The amount of users following Java naming conventions has decreased over time. Our interpretation of this result is not so much a trend towards incorrect style, but rather a trend toward a lowercase style, that may be borrowed from other languages or from a proposed wider tendency to use less capitalisation.

4.3 RQ3: do these issues of coding style and language usage change over time?

Many patterns of usage were stable over time, varying only a few percent between 2013 and 2019, which suggests that such a comparison is reasonable on this size of data set. For example, Strings were used in 43.4% of projects in 2013 and 43.6% in 2019; if-statements

were used in 44.9% of 2013 projects and 38.4% of 2019 projects. We have no particular reason to expect changes in such fundamental aspects of the language. This gives us a basis to examine some of the other changes, such as the change in main method mentioned in the previous section.

Usage of built-in data types was generally stable over time, but the use of other types varied more widely. We detected a possible change in programming tasks, with GUI classes declining in BlueJ between 2013 and 2019, while text-parsing classes (e.g. Scanner) increased during the same time. It may be that educators have moved GUI programming from the first course (where BlueJ is most often used) to later courses (where they may move on from BlueJ to another Java IDE, like IntelliJ) and hence what looks like a decline in use is actually a postponement in use. Another hypothesis is that GUIs are increasingly being covered in Web development courses.

The use of the for loop increased, and the use of the while loop decreased. We hypothesise that the introduction and the subsequent teaching of "for each" style loops in Java 5 contributed to this phenomenon, rather than our initial hypothesis of a longstanding movement away from Iterator (the latter has equally been observed between 2013 and 2019).

A noticeable shift was found in exception behaviour. The use of try-catch and throw roughly halved between 2013 and 2019, while the use of throws (indicating a method may propagate the given exception) increased. This suggests users are writing less exception-handling code, in favour of propagating them out to other methods and/or not catching them at all. The use of IOException in catch blocks reduced by around 80% from 2013 to 2019; this may partly relate to an increase of use in the Scanner class, in which the reading methods do not throw any IOException.

There was also a big reduction in the use of static methods and fields from 2013 and 2019. We believe this was primarily driven by the reduction in use of the static main method, but it may also be a sign that instructors who previously used static as a way to "escape" the strongly object-oriented nature of Java have either moved to true object-orientation, or moved on from Java (e.g. to Python).

5 LIMITATIONS

There are potential confounds with the 2013 data set. This data set was collected at the launch of Blackbox. Users would appear in Blackbox for the first time after they started using BlueJ 3.1.0. However, there is no way to distinguish between first-time users and pre-existing users who upgraded. Therefore it is possible that users in the 2013 data set had more experience than users in the 2019 data set, although neither 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 less files per project on average, suggesting that this may not be an issue.

A standard caveat of Blackbox research is that Blackbox, and therefore Blackbox Mini, does not contain user-profile information. We are thus unable to know the background of BlueJ users, such as their experience with other programming languages, or their current programming task. However, for the analyses in this paper, this is not a particular limitation as we were interesting in programming use "in the wild" in BlueJ without regard to institution or course or a specific programming task.

The analyses here all look at syntactic information, rather than a full semantic analysis of programs. For example, our analysis of data type use looks at variable declarations, rather than implicit use of types within expressions. So code with an intermediate variable might show a data type use that code with method chaining would not. We believe that in terms of analysis effort, syntactic analysis is an order of magnitude simpler than a full semantic analysis of a program, and thus where syntactic analyses can be used as a reasonable proxy for a fuller analysis, this is wise and justifiable.

6 CONCLUSION

We examined three research questions in this paper:

- We found that BlueJ users do not use all of the language, and in particular the number of built-in data types they use is small. We suggest that beginner languages only need one integral type, one floating point type, a boolean and a string type built-in. However, use of other types varies widely, and BlueJ users make use of many different library classes.
- We found that although the BlueJ tool promotes a certain style of coding, this does not seem to have a strong effect on users, who are presumably instead influenced by their learning materials, instructor, and prior experiences. Passive promotion of coding practices in the tool does not seem to be a strong factor.
- We found that many coding patterns are stable over time in this size of data set (roughly ten thousand projects in each of 2013 and 2019). The main shifts were found in use of syntax constructs and data types, particularly a change in exception behaviour and the use of static methods and fields. The usage of library data types changed; we believe this was because of a shift in the type of programming assignment being performed.

Our analyses were performed using small analyses on code stored in the srcML format. Our experience was that this XML format makes adding extra analyses straightforward, and we believe that this data format (and the Blackbox Mini data set that uses it) is a promising avenue for source code analysis in programming education research. An example analysis is given in Figure 1 and uses under 20 lines of Python to count the number of source files in a project with a Javadoc comment. Our other analyses are similar in terms of code complexity.

An obvious avenue for our future work is to use further slices taken from intervening years, both to improve the validity of our analysis through increased sample size, and to check whether the patterns over time are stable trends.

ACKNOWLEDGMENTS

We are grateful to Jonathan Maletic for introducing us to SrcML and to King's College London for their undergraduate research fellowship scheme which supported one of the authors. We are grateful to Charalampos Kyfonidis and Michael Kölling for their comments on the draft.

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* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 53–61. https://doi.org/10.1145/2960310.2960325

[3] 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* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 210–214. https://doi.org/10.1109/MSR.2019.00042

[4] 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/

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

[6] 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

[7] Neil C. C. Brown, Amjad Altadmri, Sue Sentance, and Michael Kölling. 2018. Blackbox, Five Years On: An Evaluation of a Large-Scale Programming Data Collection Project. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 196–204. https://doi.org/10.1145/3230977.3230991

[8] Neil C. C. Brown and Michael Kölling. 2020. Blackbox Mini - Getting Started With Blackbox Data Analysis. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 1387. https://doi.org/10.1145/3328778.3367006

[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* (Atlanta, Georgia, USA) *(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] 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

[12] 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* (Brisbane, Queensland, Australia) *(ACE '18)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/3160489.3160500

[13] 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.

[14] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press.

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

[16] 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* (Reno, Navada, USA) *(SIGCSE '03)*. Association for Computing Machinery, New York, NY, USA, 153–156. https://doi.org/10.1145/611892.611956

[17] 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

[18] 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* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 110–115. https://doi.org/10.1145/3059009.3059061

[19] 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

[20] 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* (Darmstadt, Germany) *(ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 168–172. https://doi.org/10.1145/1999747.1999796

[21] 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.

[22] 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

[23] P. Techapalokul and E. 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.

[24] 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.

[25] 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* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, 13–18. https://doi.org/10.1109/ICPC.2019.00015