Do Java Programmers Write Better Python? Studying Off-Language Code Quality on GitHub

Siegfried Horschig
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
siegfried.horschig@student.hpi.de

Toni Mattis
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

There are style guides and best practices for many programming languages. Their goal is to promote uniformity and readability of code, consequentially reducing the chance of errors.

While programmers who are frequently using the same programming language tend to internalize most of its best practices eventually, little is known about what happens when they casually switch languages and write code in a less familiar language. Insights into the factors that lead to coding convention violations could help to improve tutorials for programmers switching languages, make teachers aware of mistakes they might expect depending on what language students have been using before, or influence the order in which programming languages are taught.

To approach this question, we make use of a large-scale data set representing a major part of the open source development activity happening on GitHub. In this data set, we search for Java and C++ programmers that occasionally program Python and study their Python code quality using a lint tool.

Comparing their defect rates to those from Python programmers reveals significant effects in both directions: We observe that some of Python's best practices have more widespread adoption among Java and C++ programmers than Python experts. At the same time, python-specific coding conventions, especially indentation, scoping, and the use of semicolons, are violated more frequently.

We conclude that programming off-language is not generally associated with better or worse code quality, but individual coding conventions are violated more or less frequently depending on whether they are more universal or language-specific. We intend to motivate a discussion and more research on what causes these effects, how we can mitigate or use them for good, and which related effects can be studied using the presented data set.

<Programming'18> Companion, April 9–12, 2018, Nice, France

ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3214341

CCS CONCEPTS

• **Social and professional topics** → **Quality assurance**; Software engineering education; • **General and reference** → *Metrics*; • **Software and its engineering** → Software defect analysis;

KEYWORDS

code quality, best practices, lint, github, explorative study

ACM Reference Format:
Siegfried Horschig, Toni Mattis, and Robert Hirschfeld. 2018. Do Java Programmers Write Better Python? Studying Off-Language Code Quality on GitHub. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3191697.3214341

1 INTRODUCTION

While writing programs in a modern programming language, programmers are confronted with numerous ways to obtain the same behavior. Apart from algorithmic details and technical trade-offs, *code style* plays an important role in selecting how a particular concern is being implemented. For example, a programmer might ponder on whether a data record justifies introducing a class or if a tuple or dictionary is sufficient, whether to use accessor methods instead of public fields, or whether to capitalize a name.

Style guidelines and *coding conventions* have been established by most programming language communities to support such decisions and maintain a uniform appearance of source code. Adhering to guidelines promotes readability and, as a consequence, reduces the chance for errors. In collaborative settings, best practices lower barriers by minimizing surprises and maximizing recognizability for potential contributors.

Problem statement. We assume that programmers which regularly use a language will eventually learn and internalize most coding conventions. However, little is known about what happens when they *switch* languages and write code in a language that is not their "every-day language".

Insights about factors leading to coding convention violations could help to improve tutorials for programmers switching languages and give rise to tools that adapt to programmers' main language. In the context of teaching programming, this knowledge might influence the order in which programming languages are taught, or make teachers aware which errors they will likely encounter given the background of their students. Conventions that are frequently violated can serve as counter-examples for language designers.

The opposite effect, i.e., off-language programmers being unexpectedly better at a casually used language than programmers with that particular language as their primary language, can help to understand which concepts or conventions of their primary language transfer well to the casually used language.

A GitHub-based study. Collaboration platforms like GitHub[1] publicly host open source development processes, which provide large amounts of data that can help to identify programmers that are likely familiar with one language and occasionally develop in another language. The off-language projects primarily maintained by them can then be downloaded and analyzed for compliance with coding conventions using existing analyzing tools, so called *linters*.

In the scope of this work, we define thresholds (as lines of code contributed in the respective languages) that identify our subjects among more than 14 million GitHub users and select their projects from the total of about 40 million repositories. For further analysis, we will restrict ourselves to the programming languages Python, Java, and C++, and specifically analyze Python design defects caused by programmers of all three languages.

1.1 Results

When reporting the defect rate per line of code, several defect classes show statistically significant deviations depending on what primary language has been used.

We observe that conventions which are more relevant in the Python ecosystem are violated more frequently, especially those regarding the 80-character line length, indentation, scoping, and formatting of if/else-blocks. We found evidence that some obvious mistakes are in fact more common, such as leaving semicolons at the end of a Python line.

Surprisingly, C++ programmers tend to use undefined variables, not use defined variables, and shadow variables from outer scopes much more often than Python programmers do.

Java and C++ programmers are better than Python-only programmers at avoiding too large or too small classes (but not necessarily complex methods), which may be caused by their experience in object-oriented design that can be transferred to the Python world. Also, formatting expressions with sufficient whitespace and grouping import statements seems more intuitive for Java/C++ programmers.

2 APPROACH

2.1 A GitHub Data Set

A large part of today's open source development is hosted by the GitHub platform. Approximately 15 million users and organizational accounts maintaining 40 million public software repositories accumulated more than 500 million versions[1], i.e., Git commits.

GitHub provides an HTTP-based API that can be queried for JSON data on any artifact (user profile, repository, commit, etc.), the possibility to clone repositories including their full version history using Git, and the newest version of each repository as non-versioned ZIP archive for download. The fact that this data is public and relatively accessible by standard web tools allows quantitative studies of programming activities backed by large amounts of real-world data.

Mirroring public GitHub. The *GHTorrent* project mirrors the public GitHub timeline, collecting commits, projects, user profiles, issues, and other related artifacts using the API [2]. They maintain two databases, a smaller relational database with pre-processed and normalized meta data, and a document-based database with the full JSON replies returned from the GitHub API, including commit contents. They provide CSV-formatted snapshots of the relational schema compatible with MySQL as well as BSON-formatted snapshots of the document-based schema meant to be loaded by MongoDB.

Our setup is based on a relational database snapshot[2] imported into a PostgreSQL schema. Since this normalized schema is missing most memory-intensive textual data (commit messages, file paths, file patches), we imported additional commit data, such as commit message, file names, and the serialized diff from the document-based snapshot into the relational schema.

Merging both data sources was done by traversing the MongoDB database dumps using the `libbson` library and inserting the information depicted in the right part of Figure 1 via a PostgreSQL connector and the `COPY` bulk-loading facilities of the database. This process required approximately three weeks and discovered multiple cases of invalid unicode, resulting in the field being skipped. Another week was needed to remove exact duplicates and create indexes. Surprisingly, we found about 50% more commits in the imported table than in the relational GHTorrent schema.

2.2 Finding Candidate Users

Our first goal is to identify GitHub user profiles that match our notion of having contributed a lot in one programming language and a little in another language, e.g., 200 LOC Python code compared to 10,000 LOC Java code. We also require them to not have a significant contribution in one of the remaining languages under consideration[3] to better attribute effects to the primary language, e.g. considering Java and C++ as primary languages, we would not want programmers that have profound experience in both of them.

Our selection consists of two thresholds: the minimum number l_{min} of lines of code required to be a secondary (casual) language and a multiplier m_{prim} how much more LOC users has to have contributed in their primary language. That means, given we sort each users' LOC counts per language in descending order $(l_1, l_2, \ldots)$ we look for all users matching all of the following criteria:

(1) $l_2 \geq l_{min}$ LOC in their secondary language,
(2) $l_1 \geq l_2 \times m_{prim}$ LOC in their primary language,
(3) and $l_n < l_{min}$ in any other language $n > 2$.

Counting LOC. Since each commit retrieved from GitHub contains the number of lines changed (i.e., overwritten or added) per file, we can attribute all files with the respective language-specific extension (e.g. `.py` in Python, `.java` in Java, `.cpp, .hpp` in C++) to a user/language combination and sum over the lines changed. This does not include deletions.

[1] before 2017

[2] Snapshot timestamp: 2017-01-01

[3] Including *all* languages known to us would require distinguishing thousands of languages in 8 billion contributions

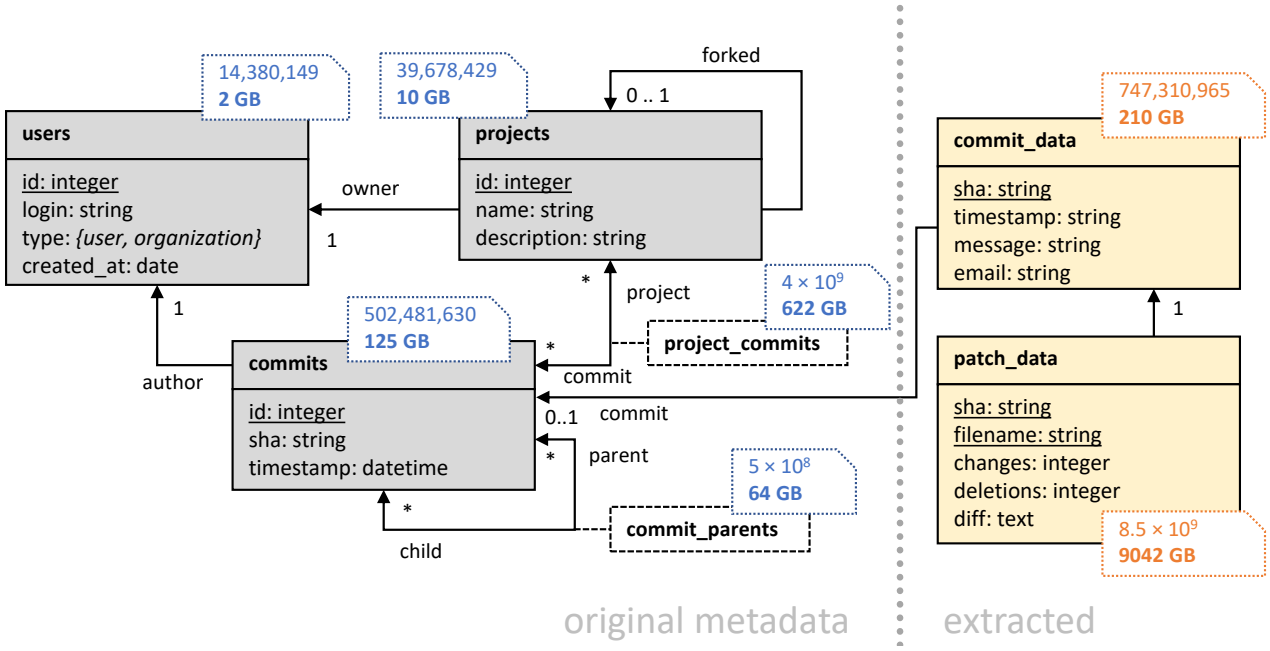

Figure 1: Relevant parts of our modified GitHub data set. GHTorrent relational data on the left side, additions extracted from GHTorrent's document database on the right. n-to-m relations are realized by join tables (broken line). Number of rows and total size (including indexes) on disk are reported in dotted boxes, the large size of join tables is mostly caused by indexes from both sides. Relations not depicted here include issues, pull requests, milestones, and user comments.

Control group. We also identify users that only contributed in the language in which we measure code quality. We intend to use their quality metrics as a baseline to compare off-language contributors to.

A requirement for control group members is that they must have contributed at least $l_{min} \times m_{prim}$ LOC in that language and less than l_{min} LOC in any of the other candidate groups' primary languages considered here. That means that the control group has no secondary language.

To keep the studied groups of programmers comparable, we will randomly select approximately as many control group members as we have in each of the groups for the other primary languages.

Setting the minimum LOC l_{min}. Probing different values of l_{min} (from 10 to 1000 in steps of 10) fixing $m_{prim} = 1$ and counting the number of users fulfilling the above criteria yields an optimum around $l_{min} = 150$ LOC (see Figure 2). For lower thresholds, we observe that programmers are more likely to have a third significant language and violate condition (3), for higher thresholds we find less programmers having a secondary language according to condition (1) at all.

Setting the factor m_{prim}. Ideally, we want to find experts in one language having orders of magnitude more contributions compared to their second language to see the clearest effect. However, the number of candidates exhibiting a usage factor between their first and second most used language already drops sharply from

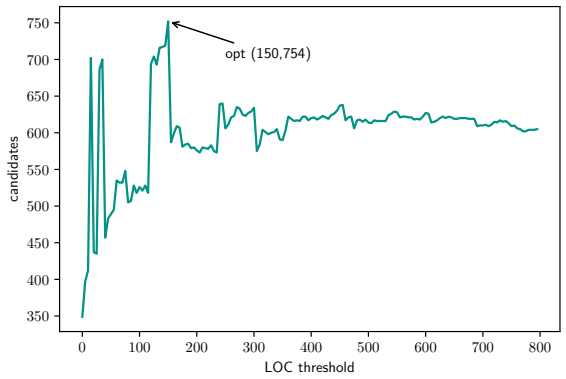

Figure 2: Number of candidates for different values of the minimum LOC l_{min}.

$m_{prim} > 1$ (see Figure 3), so we fixed $m_{prim} = 5$ for now to trade candidate numbers against expertise difference.

2.3 Finding Candidate Projects

After agreeing on a set of primary and secondary languages, we collect all projects contributed to by users whose primary language is within our scope, and select those that received contributions by the users in their second language.

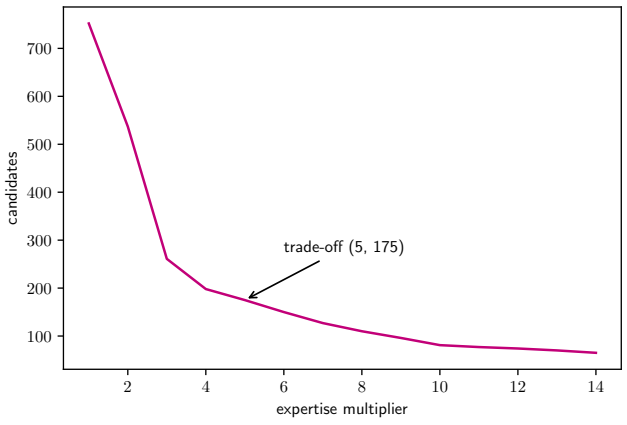

Figure 3: Number of candidates for different experience-gap multipliers m_{prim}.

For example, in the study presented in the remainder of this paper, we will select projects edited by *Java* and *C++* programmers that are written in *Python* themselves, i.e., Python is going to be the secondary ("switch to") language and Java and C++ the primary ("switch from") languages.

Additionally we select the projects contributed to by our control group to measure the code quality baseline. In our case, these will be Python projects edited by Python programmers.

2.4 Measuring Violations of Coding Conventions

To measure code quality in terms of compliance with coding conventions and best practices, we propose to use *linters* for the respective secondary language. A linter apply a series of checks to a program, reporting locations where code quality may be compromised. We can count the prevalence of reported locations as an indicator for code quality.

PyLint. An example for a widely used linter is the *Pylint* software for Python [4]. Pylint can report the following message types, associated with the file and line number where the problem occurred:

Fatal errors prevent Pylint from further processing the file (e.g., unbalanced brackets),

Errors render the Python module invalid (e.g. duplicate argument names) or likely causes a runtime error when the reported line of code is executed (e.g. variable use before assignment),

Warnings are emitted for error-prone code or style issues (e.g., unreachable code, mutable default values in Python, overly general exception handling),

Refactoring hints indicate code that is too complex or violates modularity (e.g. code that is nested too deeply, boolean expressions that can be simplified),

Convention issues refer mostly to formatting and naming (e.g., wrong number of spaces around an operator).

Each message has a unique ID, which is prefixed with either F, E, W, R, or C depending on message type, followed by four digits. A short string identifying the problem is typically provided in addition to the code, e.g. the error `E0110 abstract-class-instantiated`.

For each candidate project, we collect location and message ID for each occurrence of a message reported by the linter.

Defect Rates. For each project analyzed by the linter, we determine a relative rate for each of the reported message IDs. That is the number of times we encountered this particular message divided by the total number of lines reported by the linter.

For each message ID, we obtain a set of per-project data points between 0 and 1 for each primary language and one for the control group. These sets capture *inter-project variability* in defect rates.

Statistical Significance. To compare such a set of off-language defect rates to the control group, we use the Wilcoxon-Mann-Whitney-Test against the null hypothesis that the off-language defect rates are a sample from the control group's distribution.

This test is done for each primary language and each message ID and we reject the null hypothesis at $p < 0.05$. We drop any message ID from the results that shows no significant difference.

3 RESULTS

3.1 Data Breakdown

In this analysis, we only used Python as secondary language (both 2.x and 3.x versions) and Java and C++ as primary languages. From the approximately 18 000 users that had contributions in Python and one of the other languages, we selected:

- All 84 Java candidate users meeting the criteria described above,
  - all 45 Python projects edited by those users, of which
  - 40 Python projects were actually reachable at the time of download from GitHub.
  - In total 2363 source files with 480 875 LOC[4].
- All 91 C++ candidates,
  - all 41 Python projects edited by them, of which
  - 33 Python projects were reachable.
  - In total 661 source files with 175 402 LOC.
- 100 Python control-group candidates from 1 800 qualified Python experts,
  - all 420 projects edited by them, of which
  - 380 were reachable.
  - In total 12 197 files with 1 335 220 LOC.

On the one hand, the candidate users making up only 0.0019% of GitHub users in total might be regarded too few for a representative sample, on the other hand the required minimum experience and experience gap between two languages is a very strict precondition. If we cannot demonstrate effects in this group, relaxing the requirement will unlikely lead to more salient results without switching to sophisticated statistical models.

3.2 Code Quality Issues

We report on a selection of defects that were significantly more or less prevalent in the Java and C++ candidate groups compared to the Python control group. The summary in Table 1 reports the

[4]Lines of code **excluding** comments and blank lines

Table 1: Ratio of defect rates compared to the Python control group.

Code quality issue	Java group	C++ group
long line	3.59	1.44
invalid name	1.43	1.52
wrong import order	–	1.83
ungrouped imports	0.16	0.14
bad whitespace	–	0.38
semicolon	4.42	20.62
redefined builtin	0.57	–
bad intentation	3.39	3.28
redefined outer name	1.68	2.21
undefined loop variable	–	3.28
unused import	0.63	0.81
unused variable	1.56	2.25
complex function	0.84	1.48
too many public methods	0.26	0.46
too few public methods	0.34	0.58
no else return	–	1.52
undefined variable	–	1.55
assignment from no return	28.27	–

comparison as quotient between average defect rate in Java/C++ candidates' projects and average control group defect rate. Values less than 1.0 indicate an improvement in defect rate. Values that did not pass the significance test ($p \geq 0.05$) are not reported in the table.

Line too long (C0301). In Python, lines should not exceed 80 characters in length. Both C++ and Java candidates have higher rates of too long lines ($p < 0.001$) in Python.

Invalid name (C0103). Naming conventions in Python state that, e.g., class names are capitalized while method and field names start with lowercase letters. Programmers from both primary languages tend to violate naming conventions almost one and a half times more often than Python programmers ($p < 0.001$).

Wrong import order (C0411). Module imports in Python are ordered so that standard library goes first, followed by third-party libraries, and eventually local imports.

C++ programmers violate this convention 83% more likely than Python programmers ($p < 0.01$), while Java programmers show no difference at all.

Ungrouped imports (C0412). In Python, multiple import statements from the same package should be grouped together. Java and C++ programmers group their imports much more often than our Python control group, violating this convention only at 16% and 14% of the control group's rate respectively ($p < 0.001$).

Bad whitespace (C0326). C++ programmers have a 62% lower chance of missing or placing too much whitespace around operators, brackets, or blocks ($p < 0.05$). Java programmers also seem to use whitespace more reliably than Python programmers, however effects where not significant.

Unnecessary Semicolon (W0301). It does not come as a surprise that programmers from C-like syntax tend to use semicolons in other languages by mistake. In Python, a semicolon is a logical line delimiter having no effect when put at the end of a line. C++ programmers are 20 times more likely to introduce an unnecessary semicolon ($p < 0.0001$). Java programmers have a 4 times higher density of semicolons ($p < 0.0001$).

Redefining built-in names (W0622). Especially when not familiar with Python, there is a tendency to name things the same way as those in the built-in namespace. Typical examples include naming a variable `input`, which is the built-in function to read a line from terminal, or naming a string argument `str`, which is the name of the string type. This may lead to unexpected or confusing errors when the built-in construct is later used the way it was intended.

One might assume that being unfamiliar with Python increases the chance, but Java programmers show about half the frequency of this issue ($p < 0.01$). We believe that Java programmers are used to more sophisticated programming environments and will likely use one to write Python as well. In general, such environments highlight built-ins in a way that accidental use of them would become apparent immediately.

Bad indentation (W0311). Since Python has indentation semantics, using indentation consistently is crucial. Swapping spaces for tabs or using a different number of spaces at different code blocks can still have unambiguous semantics, but is considered bad style. Both Java and C++ Programmers are about 3.3 times more likely to introduce inconsistent indentation ($p < 0.05$).

Redefining outer name (W0621). Shadowing a name from an outer scope is a common cause of errors in Python and thereby discouraged.

Java programmers are 58% more likely to redefine an outer name ($p < 0.01$), C++ programmers are 2 times as likely ($p < 0.01$).

Undefined loop variable (W0631). Using a loop variable outside the loop can be useful to inspect the value it had after the last loop run. However, when the loop never ran, this variable is undefined.

C++ programmers are 3 times more likely to use loop variables outside the loop ($p < 0.05$). An influence might be the fact that in C++ that follows C practices, loop variables are often declared at the start of a function, making them available before and after a loop and.

Unused import (W0611). C++ programmers are 24% less likely to import something they do not use within the module ($p < 0.05$). However, since imports are at file level and not something anyone would consider doing *per line*, this statistic is questionable. Typically, the amount of imports does not scale linearly with the length of a file, hence a redundant import in a smaller file has much higher impact on the statistic than one in a larger file.

Unused variable (W0612). Java programmers tend to forget about a previously defined variable 55% more often than the control group ($p < 0.05$), C++ programmers have a 124% higher rate of this particular warning ($p < 0.0001$).

Complex method/function (R1260). PyLint considers a method or function too complex when its cyclomatic complexity exceeds 10.

This happens to C++ programmers 67% more often than to Python programmers ($p < 0.05$). The Java group had approximately the same complexity ratings as the control group.

Too many public methods (R0904). This message is emitted when classes have more than 20 public methods. Java and C++ programmers tend to make smaller classes, thereby producing this refactoring hint at less than half the rate of Python programmers ($p < 0.0001$).

Too few public methods (R0903). This code smell indicates a class has been created only for the purpose of storing data. In such a case, Python programmers should use simpler structures, e.g., tuples, dictionaries, or instances of the `namedtuple` meta-class. Alternatively, this can be a symptom of state processing happening somewhere else, when it should be a responsibility of the class.

Both Java and C++ programmers are *less* likely to produce data classes ($p < 0.001$) in Python. Both languages enforce object-oriented design in a stricter way than Python does. Especially in Java, programmers need to decide which class their method belongs to, while Python programmers can simply put a function at module level outside a class (in Java, a comparable design would require making a method *static*).

No else return (R1705). In Python, having an `else` statement right after an `if`-branch returns is considered bad style as it increases complexity. C++ candidates tend to use the redundant else branch more often ($p < 0.05$). The example below illustrates a violation of this coding convention:

Listing 1: The second return statement should not be nested in a redundand else branch.

```
if condition:
    return a
else:
    return b
```

3.2.1 Errors.

Undefined variable (E0602). Using a potentially undefined variable is a common phenomenon, even Python programmers do this in about 6 out of 1000 LOC. Often, these are situations not reachable by the intended control flow, but during program evolution these control flows may emerge and lead to errors later on.

C++ programmers have a 55% higher rate of using undefined variables ($p < 0.001$). Java programmers are about 20% less likely, but high variance makes the result insignificant.

Assignment from no return (E1111). Java candidates sometimes use functions without return value in an assignment or as an expression ($p < 0.05$). In Python, such a function would return `None`. Especially, if only some code paths return a value and the program is fine with the `None` result in some cases (e.g. because it would be interpreted as false in a boolean expression), the error might remain undetected at run-time, while compile-time checks would have found the mistake in Java. In C++, this would be undefined behavior and not necessarily caught by compilers, so C++ programmers might be wary here.

Listing 2: Missing the last `return` is bad practice, but has no effect if the function is used as condition.

```
def is_square(rectangle):
    if rectangle.a == rectangle.b:
        return True

if is_square(some_rectangle):
    # ...
```

4 DISCUSSION AND FUTURE WORK

We identified a wide range of *threats to validity* and, as a direct consequence, propose a number solutions that would improve confidence in the results. Besides that, we will discuss interesting follow-up questions resulting from this initial study.

4.1 The GitHub Data Set

Studies like this demonstrate the capabilities of having a comprehensive GitHub data set at hand.

The study also showed several weaknesses of the data set: A re-occurring problem is keeping a large database up-to-date. The longer we used the 2017 snapshot, the more projects became unreachable because they have been relocated (e.g. from a user account to an organization), removed, or renamed, or their owner's profile does not exist anymore. As another side effect, our candidate selection is based on commit data that was available through all of 2017, while the linting results are done on more recent project versions. Original contributions we were hoping to measure could be overwritten in the meantime. This should be addressed in future studies by doing a fully version-controlled "Git clone" of each project, and subsequently rolling back to the last version that the currently loaded GHTorrent data set knows about.

Updating the database regularly would consume a significant amount of bandwidth, storage, and computational resources, reducing the availability to researchers, and undermining long-running analyses that assume their view on the data does not change from one day to the next. Addressing the trade-off between (transactional) data consistency needed by long-running analyses, reproducibility with original data, and replicability with up-to-date information is a challenging topic for future work.

4.2 Candidate and Project Selection

To keep the candidate selection as intuitive as possible, we accepted a number of threats to validity concerning our candidate set:

- The candidate sets are small compared to the size of GitHub's user base. Relaxing the requirements would give us more programmers at the cost of a smaller experience gap. A statistical model that would be able to report how defect rates shift depending on how many lines the programmer has written in each language could help to separate the effects of each individual language on code quality, for example via logistic regression or analysis of variance (ANOVA).
- The off-language programmers may have programmed a little bit of Python *before* they switched to Java or C++ completely. Then, their later primary language would have had

no impact on their Python skills. Using commit dates, we would need to check to which degree this might be the case.
- Programmers might have copied external Python files and dependencies into their otherwise less pythonic repository, causing a large number of LOC being attributed to a user not actually writing that much Python.
- The degree of collaboration in some projects might be high enough that most of the code is from other programmers and not the candidates themselves. Since linters operate on full projects rather than individual contributions, we could improve accuracy by comparing line numbers of reported defects to the lines that have been changed by our candidate users and remove those that were introduced by non-candidate users.

4.3 Statistics

We did not analyze source code on a finer level of granularity, as project sizes can vary by substantial amounts. For example, half of our Python code written by Java off-language programmers is from a single project (a large-scale biology lab data processing infrastructure) that would have too much impact if we reported, e.g., file-based defect rates. Now, this large project is just a very accurate data point among 44 other projects.

However, some defects were rare in some projects (less than 10 occurrences in total), or the projects were small, which means that dividing by the total number of lines gives an estimate that is highly sensitive to fluctuations in the underlying code.

Error rates might also be biased due to project-internal guidelines that differ from the default style guide checked by the linter. To address this, a per-project comparison would be needed, e.g., comparing code contributed to the same project by authors with different language background. The project selection for this study did not yield any overlapping projects between control group and Java/C++ groups, though. Locating projects with mixed-background contributors is a challenge left for future work.

4.4 Selection of Programming Languages

We considered the Python language as the currently most popular non-JavaScript dynamically typed language in comparison to the two major statically typed languages.

In one sense, we may have measured some effects that occur when switching from statically to dynamically typed programming languages. To support this hypothesis, we would need to analyze more than a single dynamically typed secondary language, e.g., including Ruby and PHP.

Furthermore, switching primary and secondary languages, e.g., quantifying Java defects caused by Python programmers, would be a complementary study that could potentially identify corresponding effects, e.g., if Java programmers violated less object-oriented best practices in Python than Python programmers, can we find evidence that Python programmers write inferior object-oriented code in Java?

By contrast, a challenge when linting languages is the C family of languages, since without running the preprocessor and having all header files ready, one cannot even guarantee balanced brackets.

4.5 Qualitative Analysis

Analyzing a considerably sized data set helps to statistically distinguish anomalies, like the difference in defect rates, form "background noise", and allows to correlate them with the primary language of the programmers. Nevertheless, this approach cannot explain why and through which process the effects manifest themselves.

4.6 Measuring Bad vs. Good Features

In this work, we only focused source code features reported by linters. According to community consensus, they are associated with code that is harder to maintain. Code quality has been measured as the absence of such *bad* features.

For future work, we propose to analyze *good* (or neutral) features as well, for example, the use of functional concepts over for-loops, patters and built-in modularity concepts over tightly coupled modules, expressive names, and other indicators of proficiency in a programming language and programming in general. This idea gives us two main directions to continue this work:

- *Measuring expertise* by used language concepts rather than lines of code. For example, Python programmers who regularly write Python for their job, but rarely contribute to open source projects, would easily slip through our LOC-based candidate filter. However, using advanced Python features in their few contributions, or improving modularity metrics rather than worsening them, can reveal that they are experts.
- *Measuring how paradigms transfer* across languages. Instead of pointing out errors made by off-language programmers, a follow-up study could try to find evidence for advanced language concepts, or specific styles (e.g., functional, object-oriented), that are recognized and readily picked up by programmers from other languages.

5 RELATED WORK

Studying development processes and effects on code quality in large scale data sets like GHTorrent is a recent development. Examples include the work from Ray et al. where effects on software quality were traced back to language design aspects, controlling for a wide range of factors [5]. Moreover, quantifying the amount of code duplicates, which is as an important aspect of code re-use and quality, has been done at large scale using a GitHub data set comparable to ours in the DéJàVu project by Lopes et al. [3].

Several interesting aspects of how programmers might misinterpret a programming language's semantics have been uncovered by Tunnell Wilson et al.[7] in an effort to crowdsource language design. Some of them do not match the actual behavior of programming languages, giving rise to errors.

One of the first quantitative studies on whether open source development differs in code quality at all and which impact modularity has on the process as well as user satisfaction has been published by Stamelos et al. [6].

6 CONCLUSION

Comparing the code quality of Python projects edited by Java or C++ programmers to those edited by Python programmers has a visible effect regarding code quality. Our data supports the assumption that

being knowledgeable in Java or C++ can actually make someone a better Python programmer regarding commonly accepted and object-oriented best practices, but not necessarily with respect to Python-specific conventions.

We also demonstrated that the GHTorrent data set can be used to study such phenomena without the need to recruit programmers for a user study, but the observations are much less reliable compared to controlled studies and require assessment of several random factors influencing code quality. While trying to minimize the room for errors, we were left with only a handful of programmers to study and still need to address severe threats to validity in our next steps.

Nevertheless, we are confident that this type of repository mining can be replicated to study a wider range of phenomena, such as programmers' readiness to pick up advanced language concepts or programming styles and in which way this is influenced by the languages they already know. We hope that such studies help to improve teaching activities and materials in the long run.

REFERENCES

[1] GitHub. 2018. Build software better, together. (2018). https://github.com

[2] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[3] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 84:1–84:28. https://doi.org/10.1145/3133908

[4] Pylint. 2018. Pylint - code analysis for Python. (2018). https://www.pylint.org

[5] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 155–165. https://doi.org/10.1145/2635868.2635922

[6] Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L. Bleris. 2002. Code quality analysis in open source software development. *Information Systems Journal* 12, 1 (Jan. 2002), 43–60. https://doi.org/10.1046/j.1365-2575.2002.00117.x

[7] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. 2017. Can We Crowdsource Language Design?. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, New York, NY, USA, 1–17. https://doi.org/10.1145/3133850.3133863