Catalog of Code Quality Defects in Introductory Programming

Anna Řechtáčková
anna.rechtackova@mail.muni.cz
Masaryk University
Brno, Czech Republic

Radek Pelánek
xpelanek@fi.muni.cz
Masaryk University
Brno, Czech Republic

Tomáš Effenberger
tomas.effenberger@mail.muni.cz
Umíme to
Brno, Czech Republic

ABSTRACT

Code quality is an important aspect of programming, as quality code is easier to maintain, and code maintenance makes up the majority of software cost. For that reason, code quality should be emphasized in programming education. Previous work has identified many code quality defects commonly made by students. However, the current state lacks a clear organization and prioritization of these defects. In this paper, we propose an organization framework for code quality defects, presenting a catalog that describes 80 defects, with a specific focus on defects frequently encountered in code by novice programmers. To determine which defects are worth pointing out to students, we conducted a survey among 72 educators, who rated the priority with which each defect should be reported to a student. These presented results serve multiple purposes: they facilitate comparison across various research studies, support the advancement of software tools, and offer inspiration for programming education.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Social and professional topics** → **CS1**.

KEYWORDS

code quality; Python; PEP8; novice programmers; teaching; automated feedback

ACM Reference Format:
Anna Řechtáčková, Radek Pelánek, and Tomáš Effenberger. 2024. Catalog of Code Quality Defects in Introductory Programming. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653638

1 INTRODUCTION

Code quality is a crucial aspect of programming. Recent systematic mapping by Keuning et al. [12] highlight the growing focus on this topic: Researchers are developing innovative tools and assignments to teach code quality, exploring perceptions of code quality among students and teachers, and examining methods to assess code quality.

Research in the field of code quality lacks standardized terminology, with terms like code quality issue, defect, or code smell being used with interchangeable or overlapping meanings. In this paper, we use the term *code quality defect* in meaning "any imperfect part of code for which it is possible to provide some feedback or advice" [7]. On the other hand, the frequently used term *code smell* typically refers to an indicator of potential problem [9], i.e., a code smell is often a symptom of some specific defect.

In recent years, multiple researchers analyzed code quality defects in the context of introductory programming. Effenberger and Pelánek [7] list 32 defects appearing in Python solutions from an online learning environment used by high-school and university students; they categorize the defects into six groups, which we expand on. Keuning et al. [11] list around 20 defects that teachers would point out in three pieces of Java code. Edwards et al. [6] report on detecting 112 different defects in several programming courses in Java (including advanced ones). For detection, they used popular Java linters, which mostly focus on easy-to-detect defects, like formatting inconsistencies and empty code blocks. Other researchers also list defects found in novice code. De Ruvo et al. [4] give 16 defects prevalent in Java code. Keuning et al. [10] identify 24 defects found in novice Java code. Aivaloglou and Hermans [1] explore a dataset of Scratch programs and mention two code quality defects they encountered often. Liu and Woo [14] list 10 defects most frequently occurring in Python programming assignments.

Overall, the current state of the field indicates the importance of defects and interest in their study but lacks clear organization. Existing research typically explores only a limited number of defects, partly determined by the specific course or learning environment. The lists presented in existing work intersect only on a small number of defects, though not all of the defects are specific to given programming languages. Some important defects are not mentioned at all (like the use of global variables or non-English identifiers).

To address this gap, we study the following three questions:

- **RQ1** What are common code quality defects (in the context of introductory programming) and how should we organize them?
- **RQ2** Do educators agree on the importance of defects? Which defects are perceived as most and least serious?
- **RQ3** How well do the currently available tools detect code quality defects? Where are the blind spots?

To address these questions, we have developed an extensive catalog of code quality defects. Table 1 gives examples of defects from the catalog; currently, the catalog contains 80 defects (language independent or specific to Python) [1]. We propose a systematic way to organize code quality defects, where the primary organization criteria are the relevant programming construct and defect type. We also rate the defects' severity, prevalence, tool support, and language independence.

ITiCSE 2024, July 8–10, 2024, Milan, Italy

ACM ISBN 979-8-4007-0600-4/24/07
https://doi.org/10.1145/3649217.3653638

[1] Available at https://github.com/adaptive-learning/iticse-2024-defects-catalog

Table 1: Examples of Defects and Their Attributes Tracked in the Catalog.

defect name	description	code example	construct	defect type	tool support	language indep.	severity	prevalence
redundant if-else	If-else statement with both branches being `return True/False`.	`if c: return True` `else: return False`	condition	simplifiable	3	1	5	3
duplicate expression	Repeated occurrences of the same (complex) expression.	`if a[n//2] % 2 == 0:` `  print(a[n//2] % 2)`	variable	duplicate code	1	1	4	3
while as for	While loop with the number of iterations known before its start.	`while i <= n:` `  print(i); i += 1`	loop	unsuited construct	1	1	4	2
misleading name	Variable name `i` or `j` used when traversing items, not indices.	`for i in text:` `  print(i)`	loop	poor name	1	1	4	2
missing docstring	A function, class, or method is missing a docstring.	`def f(x, y):` `  return x + y`	multiple	poor documentation	3	1	2	3
unnecessary parenthesis	A single item in parentheses after `if`, `return`, or another keyword.	`return(value)`	expression	poor formatting	2	0	3	3
too many arguments	The function has too many arguments.	`def move(board, x, y,` `  dx, dy, color): ...`	function	poor design	3	1	3	1
changing the control variable of a for loop	Changing a control variable at the end of the body has no effect.	`for i in range(n):` `  ...; i += 1`	loop	unused	1	0	5	1

To determine the severity of the defects, we conducted a survey among 72 educators at the Faculty of Informatics, Masaryk University (FI MU). For the survey, we selected 30 representative defects from the catalog. The survey shows that while the opinions on the severity of individual defects differ, some defects are generally considered more severe than others. Code blocks with no effect, using global variables and duplicate code blocks have the highest severity, while missing docstrings have one of the lowest. We extrapolated values from this survey to rate the severity of all of the 80 defects.

Some aspects of the catalog may still be subjective or disputable. Our aim is to clearly document our design process and provide a usable starting point for further development and research.

The presented results have multiple uses. For researchers, they provide a framework for organizing research on code quality defects (e.g., analyzing their prevalence or studying the impact of teaching tools on student learning). For tool developers, the results can help with selecting defects to implement and report. It can also inspire a structured presentation of detected defects to students, e.g., grouping related defects and prioritizing them. For educators, the catalog informs which code quality defects are suitable to discuss with students in a class on a given programming construct.

2 CATALOG OF CODE QUALITY DEFECTS

In this section, we present our proposal for organizing code quality defects. Table 1 shows an excerpt from the catalog; the entire catalog comprises 80 defects and is available online[2].

[2]Available at https://github.com/adaptive-learning/iticse-2024-defects-catalog

We give the rationale behind choosing what attributes to track for each defect and describe the meaning of their values (Sections 2.1 and 2.2). We also describe the process of populating the catalog with defects (Section 2.3).

2.1 Categorizing the Defects

We built the catalog to have the following desirable characteristics:

(1) All attributes and values should have a clear meaning and be intuitive for researchers, tool developers, and educators.
(2) Each value should be used reasonably often.
(3) The classification of the defects should be unambiguous; i.e., the values should have high inter- and intra-rater reliability.
(4) The classification of the defects should be consistent; i.e., similar defects are similarly classified, e.g., *unused variable* and *unused parameter* agree on five out of six attribute values.
(5) The attributes should be immediately useful to researchers, educators or tool developers.

We used these characteristics as soft guidelines informing the design process and author discussion.

In addition to name, description, and example, we track several other properties for each defect (see Table 1 for examples). We classify each defect by two nearly orthogonal properties: the type of the defect (e.g., *simplifiable*, *unused*) and the programming construct it concerns (e.g., conditions, variables).

Defect type is based on three key characteristics: what the defect looks like (e.g., code repetition), how it can be fixed (e.g., introducing a new code construct), and why it is desirable to fix it (e.g., succinctness and generalizability). Table 2 lists the defect types and

Table 2: Overview of the Defect Types.

defect type	description	how to fix	reasons to fix
unused	Redundant lines of code that do not influence the behavior of the program.	delete whole lines	tidiness, conciseness
simplifiable	Code that can be simplified by local changes that do not introduce a new construct.	local edits (reordering or deleting)	conciseness
unsuited construct	Code that can be made more explicit and simpler by using a different construct (e.g., operator, function, control flow statement)	local rewrite	self-descriptiveness, explicitness, possibly efficiency, robustness, ...
duplicate code	Repetition of identical or very similar code (expressions, lines, blocks).	replace by a suitable abstraction	conciseness, structuredness, modifiability
poor design	Code that is unnecessarily complex due to poor design choices.	non-trivial refactoring	traceability, structuredness
poor name	Inappropriate name for a variable, function or another construct.	rename	self-descriptiveness, explicitness, legibility
poor formatting	Inconsistent formatting or a violation of stylistic recommendations.	simple formatting edits	consistency, predictability
poor documentation	Missing, misleading, or unhelpful documentation (docstrings, annotations, comments).	add/rewrite documentation	maintainability

their description. The proposed set of eight defect types is based on the six types used in the most closely related previous research [7], which we refined to fit our considerably broader set of defects.

Most of the reasons for fixing the defects (e.g., conciseness, structuredness) also have their exact meaning and taxonomy, see [2]. We decided to use the negative ("what is bad") rather than positive ("why to fix it") formulations for the defect types since fixing a defect nearly always has multiple reasons. Moreover, the reasons have different strengths, and the strengths differ case by case. Consequently, we could not find a set of non-overlapping reasons to use directly for classification.

In some cases, the distinction between types is nuanced, yet their differentiation is useful. The *unused* and *duplicate code* can be thought of as special cases of the *simplifiable* and *poor design* types. We decided to keep them as separate types since: 1. They have a clear, intuitive meaning; 2. They were used in previous research on code quality defects [1, 7, 10]; and 3. We wanted to avoid inflating the already large *simplifiable* type.

The distinction between the *unused* and *simplifiable* can be defined as follows: If a whole line (or lines) can be deleted without additional edits, then this code is *unused*. If, on the other hand, the fix is just a selective deletion of characters within a single line, or if it requires some additional edits around the deleted lines, we consider it *simplifiable*.

The second property by which we organize the defects is the programming construct they concern. This property is nearly orthogonal to the defect types and is especially useful for preparing lectures since these frequently address specific programming constructs.

We use the following constructs: expressions, variables, conditions (including conditional statements), loops, functions, compound data structures, and modules. A few defects are relevant to multiple constructs and can occur for each in isolation (e.g., unreachable code); we assign them value *multiple*. On the other hand, if a defect *always* involves multiple constructs, we assign it the most advanced construct. For example, *changing the control variable of a for loop* involves both variables and loops. Still, the students can only commit the defect once they learn about loops, not variables, so it is preferable to list this defect under loops.

2.2 Ranking the Defects

One of the goals of the catalog is to facilitate prioritization – which defects to research, implement detectors for, or address in lectures. To this end, we assign each defect three importance rankings: language independence, severity, and prevalence. Additionally, we record how well is the detection of each defect enabled by the current tools.

Table 3 describes the values we used for assessing tool support and the importance rankings. The rankings are meant as initial broad estimates and merit further refinement.

The defects in the linked catalog are ordered by the sum of the importance rankings.

Tool support. We examined open-source Python linters available through the package installer for Python (PIP). We imposed these limitations to consider only those linters that can be used immediately and with relative ease. The selected linters were Pylint [16],

Table 3: Tool Support and Importance Rankings.

tool support	
3	reliably detected by multiple (open source) tools
2	detected only partially by common tools or detected well by a specialized tool
1	not detected by current tools or detected only partially by a specialized tool
language independence	
1	occurs (in similar form) in many programming languages
0	specific to Python (specific constructs or conventions)
severity	
5	both notifying the student and fixing the defect has high priority
4	notifying the student has high priority and fixing the defect has medium priority
3	notifying the student has medium priority but fixing the defect has low priority
2	notifying the student has low priority
1	the student should not be notified of the defect
prevalence	
3	can occur in many circumstances and does indeed occur
2	middle
1	can occur only in specific circumstances, or is committed by only a small subset of students

Flake8 [8] and its plugins, PyTA [13], EduLint [5] and Clonedigger [3]). For each linter, we inspected whether it is able to detect given defect and how well.

Language independence. We considered popular languages of different paradigms and levels of abstraction. If a defect or its close variation can occur in all of the languages, then it is language independent.

Severity. To determine the defects' severity, we conducted a survey (see Section 3). However, the survey did not cover all of the defects in the catalog. To rate the remaining defects, only the team of three authors rated the defects and we used the median of these ratings.

Prevalence. For defects that can be detected automatically, we analyzed several datasets of novice code to determine the number of occurrences, which we then transformed to our prevalence scale. For defects without detectors, we only used our experience with novice code. The current rating provides only coarse estimates; these should be clarified in future research.

Further in this paper, we will focus on severity and tool support.

2.3 Populating the Catalog

We base our catalog on the most extensive previously published catalog of code quality defects [7], which included 32 defects. We then added new defects, trying to leverage our combined experience in CSEd research and CS1 teaching to reveal the breadth of code quality defects. We purposefully included diverse defects (and only later captured the differences in the organization structure presented in previous sections). Despite that, the catalog is (and probably forever will be) incomplete.

We included language-independent and Python-specific defects, as Python is the language we use in introductory programming. We decided to aggregate some groups of defects (for example, most formatting rules violations are under *inappropriate whitespace: tabs and spaces*, *visible* or *trailing*) to keep the defects at similar levels of granularity.

3 SURVEY ON DEFECT SEVERITY

To determine the most subjective defect aspect, severity, we conducted a survey among educators at FI MU.

3.1 Design of the Survey

Out of the 80 defects in the catalog, we selected 30 for the survey. We did not use all in order to make the survey shorter and the respondents more likely to fill it in.

When selecting the defects, we first filtered out all defects specific to Python, as we wanted to target even educators who do not teach Python. Out of the 55 language-independent defects in the catalog, we selected 30 to cover all defect types and programming constructs. We also rated the defects' severity preliminarily and then selected such ones so that defects with different levels of this preliminary severity would be present.

We conducted the survey through a custom web page. It collected basic demographic information on each respondent (name, years of experience, experience with different student groups, like university or high school students). It instructed respondents to rate how severe they consider the defects for CS1 students.

For each defect, the page displayed its name, description, code example and fixed code example (same as in the linked catalog). Based on that information, the respondent was asked to rate the defect on the five-point severity scale.

For each respondent, the defects were presented in random order.

3.2 Participants

For participants, we reached out to teachers and teaching assistants of three introductory programming courses, two in Python and one in Haskell. We also contacted students and alumni of a course on university teaching (all of whom have practical experience with teaching).

We collected responses from 72 educators. Out of these, 46 had at least two years of experience in teaching and 67 had experience teaching university students of computer science (others had experience with different student groups).

All of the participants were students or employees at FI MU.

3.3 Results

The survey results are summarized in Figure 1. The results show that there are large differences in defect severity. *Block with no effect* is considered severe almost unanimously, while almost half of the participants consider *argument does not come first in comparison* not worth mentioning.

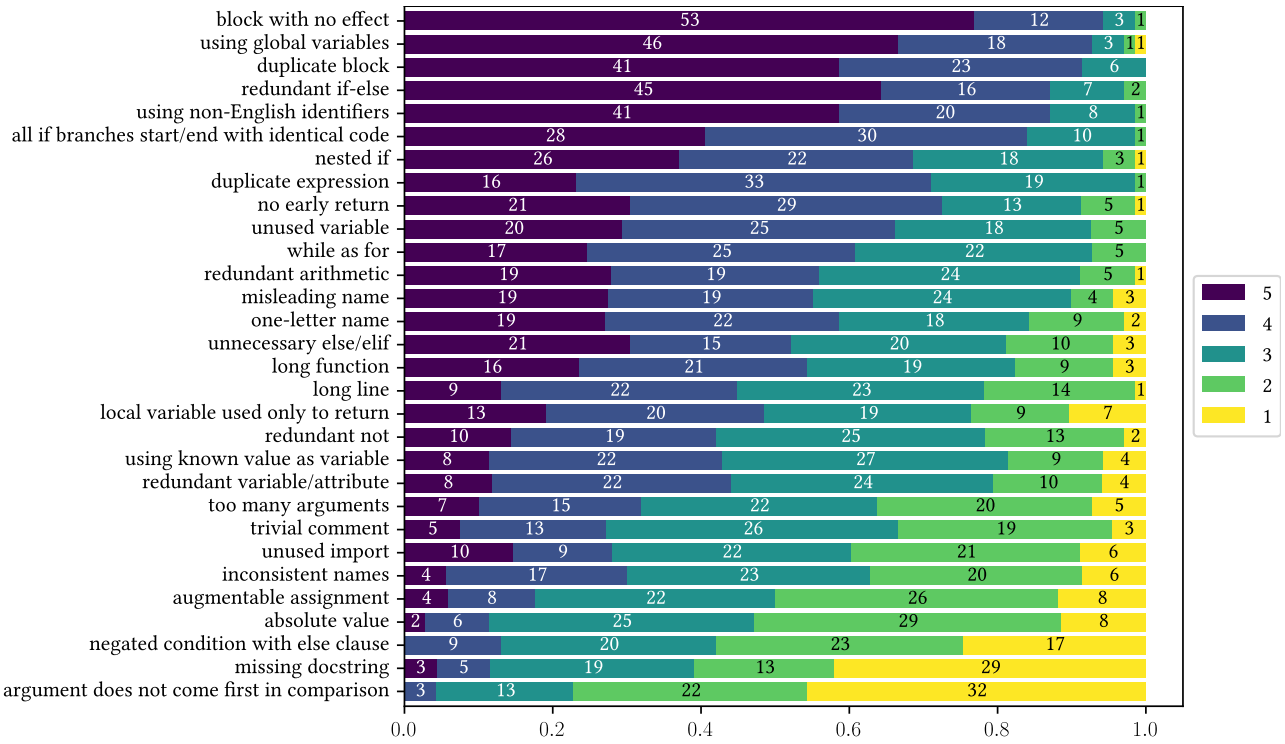

Figure 1: Ratio of Ratings for Each Defect, Sorted by Average Rating (N = 72).
The number in each bar segment gives the number of respondents who chose that option.

While individual opinions sometimes differ widely, the overall degree of agreement (measured by standard deviation) is similar (around 1) across all defects. The defect with the highest degree of agreement (0.63) is *block with no effect. Local variable used only to return*, on the other hand, exhibits the lowest degree of agreement (1.23).

A possible explanation for the differences in ratings is that different educators have a different level of strictness about what they would require from their students. To see how this would affect the results, we normalized the ratings of each educator by their average rating. Nevertheless, this made little difference in the relative order of defects. So, there may be differences in how severe different educators consider one defect, but there seems to be much higher agreement on which defects are more severe than others.

3.4 Limitations

Our survey contains several limitations, which could be beneficial to address in future research. Most importantly, the severity of a defect may vary between its occurrences (e.g., the extent of duplication for duplicate block), and it may depend on context (e.g., single letter variable name `i` used as an index or as a character in a string). Therefore, the severity rating from the survey may depend on the specific example we presented.

The formulation of severity levels is a compromise between clarity and conciseness. While designing the survey, we considered several variants of severity level formulations, and the specific choice of formulations may have impacted ratings by participants.

The defect severity can differ for different target groups; e.g., defects that are severe from the perspective of CS1 may not be as severe when teaching high school students. We instructed the respondents to consider CS1 students, assuming that the severity would just shift, but not significantly change, for different target groups. While this seems to us a reasonable assumption, it is still unverified.

All respondents of the survey were affiliated with a single institution. It is thus possible that the results are influenced by the shared background of the respondents. While we believe that the described factors do not have a fundamental impact on the presented results, it may be useful to replicate the survey in a modified setting.

4 TOOL SUPPORT

Figure 2 shows the relationship between severity and tool support for different defect types. It is necessary to keep in mind that while the catalog is extensive, it still contains only a sample of defects, and some categories contain only a few defects. Also, we evaluated tool support only for open-source Python tools, so the classification is subject to our knowledge of them.

Still, the figure shows that there are deep disparities between different defect types: *poor formatting* defects are precisely defined and easy to detect (they can usually be discovered by local inspection of the code), but not very severe. Opposed to this, *duplicate code* defects are neither well-defined nor easy to detect but are among the most severe.

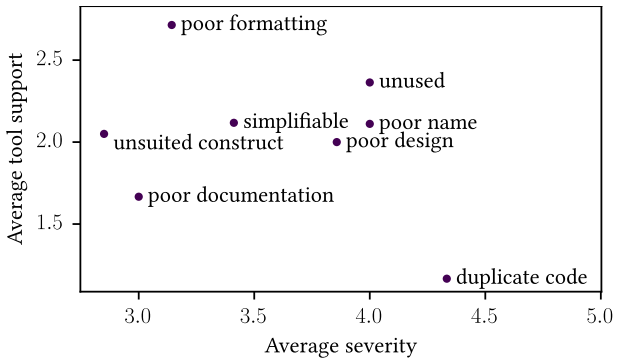

Figure 2: Defect Type Severity by Tool Support.

That is not to say there are no detectors for duplicate code. Pylint itself has a detector for duplicate code, but it only detects identical pieces of code in different files, which makes it unusable for detecting code duplication in single-file student submissions. Clonedigger reported false positives (code blocks that can not be deduplicated, like sequences of variable assignments) even for conservative parameter settings.

There are many other open-source tools for the detection of duplicate code that did not meet our requirement of being available through PIP. However, many of these are copy-paste detectors, which fail to report code where only variable names or some constants changed, which is the case in many pieces of duplicate novice code.

We further examined three clone detection tools described in a literature review by Zakeri-Nasrabadi et al. [19]: PMD [15], Simian [17] and SourcererCC [18]. For Python, PMD is able to detect identical clones only. Simian was able to detect clones differing in literals and variable names, but it failed to verify whether the segments can even be deduplicated. We were not able to detect any duplicates through SourcererCC.

Also, none of the tools we examined differentiated variations of *duplicate code*, failing to provide specific, actionable feedback to novices, who may be clueless when faced with a crude report of "duplicate code."

Duplicate code is not the only noteworthy defect type. Some severe *poor design* defects (*reimplementing a library*, *redundant variable/attribute*) are also not detected. Detecting them would require a more extensive analysis of what the program is supposed to compute, which is hard to generalize (and specific cases may not be prevalent enough to justify the effort).

Another notable defect type is *poor name*, almost evenly split between well-detected and poorly-detected defects. However, the well-detected defects all deal with the syntactic side of naming (e.g., conventions, number of characters), and most of the poorly-detected defects look at the meaning of the name.

5 DISCUSSION

We conclude with a discussion of our results, the outline of possible future work, and answers to our research questions.

5.1 Organization of Defects

RQ1: What are common code quality defects (in the context of introductory programming) and how should we organize them?

We have proposed an organization framework for code quality defects with a particular focus on novice programmers. In this framework, defects are categorized by their programming construct and defect type; for each defect, we further characterize its properties (language independence, severity, relevance to beginners, prevalence).

We also provide a publicly available catalog of 80 defects. Both the organization framework and the catalog of defects are meant as starting points for further development. The meaningful directions include:

- extending the coverage of defects,
- improving the description of defects (specifically adding more examples and tips on how to improve the code) so that the catalog can be directly used as a learning material,
- evaluating the design of the catalog, e.g., by comparing alternative organizations.

5.2 Severity of Defects

RQ2: Do educators agree on the importance of defects? Which defects are perceived as most and least serious?

We conducted a survey among seventy educators at FI MU. It shows that defect severity is at least partly subjective. However, there seems to be consensus on the relative severity of different defects.

The most severe defects are often those concerning the structure of the code (use of global variables, duplicate blocks) and those that may hint at a misconception / missing knowledge of the student (block with no effect, duplicate sequence). Defects regarding stylistic details or language-specific conventions seem to be less severe.

5.3 Tool Support and Improvement of Defect Detectors

RQ3: How well do the currently available tools detect code quality defects? Where are the blind spots?

Although many tools detect code quality defects, these tools have overlapping coverage that focuses on defects that are easy to detect automatically (e.g., formatting issues). These defects are not always the most important ones, particularly for novice programmers. Our catalog provides a clear direction for improving defect detectors: prioritize the defects with high severity and low tool support. Specific examples are detecting duplicate code and poor names. It is challenging to design detectors for these defects that achieve high accuracy. Nevertheless, it is a worthwhile effort – the task should be feasible and would be very useful in practice.

ACKNOWLEDGMENTS

We want to cordially thank all the educators at FI MU who took part in our survey. Thank you for being open to experiments, you helped us collect most interesting data!

Anna Řechtáčková is a holder of the Brno Ph.D. Talent scholarship, funded by the Brno city municipality.

REFERENCES

[1] 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*. 53–61.

[2] Barry W Boehm, John R Brown, and Myron Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*. 592–605.

[3] Peter E. Bulychev and Marius Minea. 2008. Duplicate code detection using anti-unification. In *Spring Young Researchers Colloquium on Software Engineering*. 51–54.

[4] 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*. 73–82.

[5] EduLint contributors. 2023. *EduLint*. https://github.com/GiraffeReversed/edulint

[6] Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. 2017. Investigating Static Analysis Errors in Student Java Programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 65–73. https://doi.org/10.1145/3105726.3106182

[7] Tomáš Effenberger and Radek Pelánek. 2022. Code Quality Defects Across Introductory Programming Topics. In *Proceedings of ACM Technical Symposium on Computer Science Education*. 941–947.

[8] Flake8 contributors. 2023. *Flake8*. https://github.com/PyCQA/flake8

[9] Marcel Jerzyk and Lech Madeyski. 2023. Code Smells: A Comprehensive Online Catalog and Taxonomy. In *Developments in Information and Knowledge Management Systems for Business Applications: Volume 7*. Springer, 543–576.

[10] 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*. 110–115.

[11] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 119–125.

[12] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2023. A Systematic Mapping Study of Code Quality in Education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education*. 5–11.

[13] David Liu and Andrew Petersen. 2019. Static Analyses in Python Programming Courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 666–671. https://doi.org/10.1145/3287324.3287503

[14] Xiao Liu and Gyun Woo. 2020. Applying Code Quality Detection in Online Programming Judge. In *Proceedings of the 2020 5th International Conference on Intelligent Information Technology* (Hanoi, Viet Nam) *(ICIIT '20)*. Association for Computing Machinery, New York, NY, USA, 56–60. https://doi.org/10.1145/3385209.3385226

[15] PMD contributors. 2023. *PMD - source code analyzer*. https://github.com/pmd/pmd

[16] Pylint contributors. 2023. *Pylint*. https://github.com/pylint-dev/pylint

[17] Quandary Peak Research. 2023. *Simian Similarity Analyzer*. https://simian.quandarypeak.com/

[18] SourcererCC contributors. 2023. *SourcererCC*. https://github.com/Mondego/SourcererCC

[19] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *Journal of Systems and Software* 204 (2023), 111796. https://doi.org/10.1016/j.jss.2023.111796