Introducing Code Quality at CS1 Level: Examples and Activities

Cruz Izu*
The University of Adelaide
Adelaide, Australia
cruz.izu@adelaide.edu.au

Claudio Mirolo*
University of Udine
Udine, Italy
claudio.mirolo@uniud.it

Jürgen Börstler
Blekinge Institute of Technology
Karlskrona, Sweden
jurgen.borstler@bth.se

Harold Connamacher
Case Western Reserve University
Cleveland, OH, USA
harold.connamacher@case.edu

Ryan Crosby
Durham University
Durham, England
ryan.crosby@durham.ac.uk

Richard Glassey
KTH Royal Institute of Technology
Stockholm, Sweden
glassey@kth.se

Georgiana Haldeman
Colgate University
Hamilton, NY, USA
ghaldeman@colgate.edu

Olli Kiljunen
Aalto University
Helsinki, Finland
olli.kiljunen@aalto.fi

Amruth N. Kumar
Ramapo College of New Jersey
Mahwah, NJ, USA
amruth@ramapo.edu

David Liu
University of Toronto
Toronto, Canada
david@cs.toronto.edu

Andrew Luxton-Reilly
University of Auckland
Auckland, New Zealand
andrew@cs.auckland.ac.nz

Stephanos Matsumoto
Olin College of Engineering
Needham, MA, USA
smatsumoto@olin.edu

Eduardo Carneiro de Oliveira
Utrecht University
Utrecht, Netherlands
e.carneirodeoliveira@uu.nl

Seán Russell
University College
Dublin, Ireland
sean.russell@ucd.ie

Anshul Shah
University of California, San Diego
San Diego, CA, USA
ayshah@ucsd.edu

Abstract

Characterising code quality is a challenge that was addressed by a previous ITiCSE Working Group (Börstler et al., 2017). As emerged from that study, educators, developers, and students have different perceptions of the aspects involved. The perception of code quality by CS1 students develops from the feedback they receive when submitting practical work. As a consequence of increasingly large classes and the widespread use of autograders, student code is predominantly assessed based on functional correctness, emphasising a *machine-oriented* perspective with scarce or no feedback given about *human-oriented* aspects of code quality. Such limited perception of code quality may negatively impact how students understand, create, and interact with code artefacts. Although Börstler et al. concluded that "code quality should be discussed more thoroughly in educational programs", the lack of materials and time constraints have slowed down progress in that regard.

The goal of this Working Group is to support CS1 instructors who want to introduce a broader perspective on code quality in their classroom, by providing a curated list of examples and activities suitable for novices. In order to achieve this goal, we have extracted from the CS education literature a range of examples and activities, which have then been analysed and organised in terms of code quality dimensions. We have also mapped the topics covered in those materials to existing taxonomies relevant to code quality in CS1. Based on this work, we provide: (1) a catalogue of examples that illustrates the range of quality defects that could be addressed at CS1 level; and (2) a sample set of activities devised to introduce code quality to CS1 students. These materials have the potential to help educators address the subject in more depth.

CCS Concepts

• **Social and professional topics** → **Software engineering education**; **Computer science education**; **Quality assurance**; • **General and reference** → *Evaluation.*

Keywords

CS1, code quality, examples, activities, readability, style, refactoring

*co-leader

ITiCSE-WGR 2024, Milan, Italy

ACM ISBN 979-8-4007-1208-1/24/07
https://doi.org/10.1145/3689187.3709615

ACM Reference Format:
Cruz Izu, Claudio Mirolo, Jürgen Börstler, Harold Connamacher, Ryan Crosby, Richard Glassey, Georgiana Haldeman, Olli Kiljunen, Amruth N. Kumar, David Liu, Andrew Luxton-Reilly, Stephanos Matsumoto, Eduardo Carneiro de Oliveira, Seán Russell, and Anshul Shah. 2024. Introducing Code Quality at CS1 Level: Examples and Activities. In *2024 Working Group*

Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2024), July 8–10, 2024, Milan, Italy. ACM, New York, NY, USA, 39 pages. https://doi.org/10.1145/3689187.3709615

1 Introduction

Education research regarding novice programmers has documented the difficulties that students experience when writing functional code and the misconceptions that cause incorrect code [4, 76, 150]. However, as pointed out by Gaber and Kirsh, "[a]lthough students are aware of many of their bugs and missing features, they are not sufficiently aware of the quality of their code" [52, p. 79]. Furthermore, the code quality of examples reported in textbooks has been criticised [23, 48][1], as well as several quality defects, have been found in pre-defined templates of a widespread educational platform [16].

In addition to correctness, robustness and efficiency, good code is easy to read, test and maintain. While correctness and performance can be established objectively, grading the overall quality of small programs is subjective [78]. Each individual may perceive code quality differently; for example, unlike experienced programmers, novices often find verbose code to be more readable [159]. Additionally, different groups will weight the quality criteria differently [21, 24] so that even when their perceptions of each criteria match, their overall quality assessment might differ.

Keuning et al. have recently conducted a systematic mapping study on code quality in education at all levels. They observe "much of the program quality research appeared in the last decade" [85, p. 10]. The authors conclude that "a possible direction for future work is to conduct a more in-depth literature study of" more specific topics, and "encourage researchers to perform studies on the topics that have received little attention so far, such as integrating code quality into the computing curricula, developing and evaluating course materials" [85, p. 10]. An analysis of 2020 CS1 course syllabi found that only 41 of 141 CS1 mentioned the term code quality [87]. Similarly, Börstler et al. observed that "students [...] get less information regarding code quality from their education than from other students or the Internet. [...] Giving students more exposure to issues regarding code quality as part of their education [...] could solve some of these issues" [24, p. 82].

In short, we should aspire to teach undergraduate students how to write code that, in addition to being correct and possibly efficient, exhibits *good quality*. For the purpose of this study, code quality is restricted to static properties "that can be determined by just looking at the source code, i.e., without any form of testing, or checking against specification" [139, p. 99].

This work offers a comprehensive reference for the instructor, aiming to introduce an early, yet multifaceted perspective on code quality to novice programmers. More specifically, we provide:

(i) A "catalogue" of examples that illustrate the range of quality issues affecting the typical code developed by CS1 students;
(ii) A sample set of instructional activities that is viable for CS1.

To achieve the above objectives, we have analysed examples and activities from the literature that have the potential to help educators. Moreover, we have organised the identified materials according to code quality dimensions and mapped the topics to existing taxonomies relevant to code quality in CS1. For a better appreciation of the approach to code quality taken in this study, the general perspective of the working group is outlined in subsection 2.2 and in subsection 2.4.

The paper is organised as follows. Section 2 outlines the background and our perspective on code quality in the CS1 context. In section 3, we summarise the method to review the relevant literature, to extract, select and categorise significant examples and activities, and to establish insightful connections with related taxonomies of code quality topics. The results are presented in sections 4–6 and discussed from different perspectives in section 7, where we also derive some instructional implications. Finally, we conclude by summarising the major contributions of this study and envisaging future developments.

To complete the work, a large sample of the catalogue of examples and short descriptions of selected activities on code quality are made available as appendices via an online resource [70].

2 Background

To discuss the scope of our work, this section will briefly introduce code quality as characterised by key contributions from the educational field. Note subsection 3.2 presents the method applied for the deeper and more focused literature review on the topic.

We consider code quality in the context of CS1,[2] taking into account both the type of code addressed at this level and the significant demands imposed upon novice programmers.

2.1 Programs "For People to Read"

In the mid-80s, Harold Abelson, Gerald Sussman and Julie Sussman, in the preface of the first edition of their book *"Structure and Interpretation of Computer Programs"*, described their educational approach in which programs are presented as a means of expression for humans [1, p. xxii]:

> First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read and only incidentally for computers to execute

An educational focus on the quality of programs and how programmers and learners perceive code quality can be traced back to earlier work in education research, with a similar focus on code readability. Joni and Soloway [75, p. 95] states:

> We shall argue that using efficiency as a guiding principle in critiquing working code is inappropriate for novice programmers. Instead, we develop an approach to critiquing working but poorly constructed novice programs based on the principle of *program readability*. That is, we base our critique of working code on its ability to communicate to program readers.

[1] "The object-oriented quality of many examples is not as high as one would expect to find in an introductory programming text" [23, p. 3:18]. "We found that a surprising number of resources contains at least some design smell" [48, p. 507].

[2] In this work, we use the term CS1 to mean an introductory course on programming taught in computing programs.

As development teams have become larger and code bases more complex, the importance of code comprehension has similarly grown. Fowler and Beck [50] expresses this succinctly:

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Readability and comprehensibility are crucial for code to be reusable and maintainable. If we cannot clearly understand what the code is meant to do, we cannot reuse it properly. We regularly face changing programs after they have been written because they need to be fixed or extended. Code that is easy for humans to understand is easier to modify and more likely to be correctly modified. In short, first and foremost, code quality is linked to how easy it is for a programmer to read and comprehend that piece of code. In the following subsection, we will summarise noteworthy attempts to characterise code quality more accurately.

2.2 Characterising Code Quality

High-quality code should be correct, efficient, clear and easy to maintain. While correctness and efficiency can be measured objectively, code clarity is subjective and difficult to assess. In particular, the structural characteristics of code quality are tightly connected with the quality of *design*, which inspired Waguespack's [151] reflections from an interesting philosophical perspective. His line of analysis led to a high-level categorisation presented in [152, 153], spanning far beyond the technical stance, and according to his view [152]:

> We will never be able to absolutely define design quality because of the relativistic nature of satisfaction in the observer experience. But, our students must still face design choices. So, as [Information Systems] educators we must provide a framework for them to develop and refine their individual perceptions and understanding of systems quality.

The terms "readability" and "comprehensibility", highlighted in Section 2.1, emerge explicitly from the analysis of Börstler et al.'s working group [24], who investigated how code quality is *perceived* amongst students, educators, and professional developers. In that study, they collected 34 interviews and used grounded theory to extract the quality categories in Table 1. Ranked by coded frequency, the top three were *Readability*, *Structure* and *Comprehensibility*. In particular, *Readability* was ranked first by students and developers and second by educators, who deemed *Structure* as the most important quality factor. However, we should also note that *Readability* and *Structure* are not independent criteria, as well-structured code is usually more readable.

On the other hand, readability is clearly affected by the expertise of the code reader. For example, in an early study comparing students' preferences relative to different looping strategies, Soloway et al. [137] found evidence that more compact constructs are cognitively more demanding than longer, more explicit constructs. Wiese et al. [160] pointed out that novice programmers may not understand the importance of concise structures that may make it difficult for them to read the code; in similar cases, students are unlikely to adopt a cleaner style. Similarly to Soloway, Wiese and colleagues reported in another paper [159] that students found novices' programs to be more readable than their experts' counterparts (they did agree, however, that the experts' code had better style). From a different perspective, Stoecklin et al. [141] suggested readability could be improved by using method names that meaningfully describe their purposes and by writing compelling comments, emphasising the role of good documentation rather than the code structure.

Table 1: Categories of code quality, as per Börstler et al. [24]

Category	Terms used to describe code quality
Readability	readable, no useless code, brevity/conciseness, formatting/layout, style, indentation, naming convention
Structure	well structured, modular, cohesion, low coupling, no duplication, decomposition
Testability	testable, test coverage, automated tests
Dynamic behaviour	robust, good performance, secure
Comprehensibility	understandable, clear purpose
Correctness	runnable/free of bugs, language choice, functionally correct (meeting business requirements)
Documentation	documented, commented
Maintainability	maintainable, adaptable, reusable, used by others, interoperable, portable
Miscellaneous	license, suitable data structure, metrics/measurements

In a recently published paper, Kirk et al. [88] focus on defining "code style principles" at a higher abstraction level than previous guidelines. The authors aim to build "a basis for presenting to students the key concepts for understanding and changing code" [88, p. 134]. This perspective will be discussed further in later sections.

2.2.1 Human-centred perspective on code quality. As pointed out in Section 2.1, programs are written for two audiences — a machine must execute them, but they also need to be read and understood by other programmers. As emerged from the discussion of key quality criteria within the working group, we can distinguish the following orientations toward code quality:

(i) *Machine-facing* – Regarding features that are related to program execution, e.g., correctness, robustness, security, efficiency/performance, compliance with varied user requirements;
(ii) *Human-facing* – Regarding features related to how humans interact with code, when reading, writing, and updating an artefact, e.g., readability and comprehensibility, which are facilitated by layout, formatting, naming and documentation;
(iii) *Both human- and machine-facing* – Regarding features that have an impact on both the human and the machine sides, e.g., code structure, testability, maintainability.

From our perspective, the key quality attributes pertain to human-facing orientations (ii) and (iii). In particular, by referring to Börstler et al.'s taxonomy [24], readability is facilitated by clear layouts, consistent formatting and naming conventions, whereas comprehensibility goes deeper into understanding the purpose of code. However, as a major goal of reading code is comprehending it, the terms are sometimes interchangeable in the literature. Thus, for simplicity, in what follows, we will use the single term *readability* to capture both superficial and in-depth (for comprehension/understanding) reading.

2.3 Improving the Quality of Code

A renewed interest in code quality was sparked in the mid-1990s by the work of Fowler and Beck, aimed at devising specific techniques to identify and improve poor or undesirable program patterns. To pursue this goal, they introduced the terms *refactoring* and *code smell* (the latter due to K. Beck), which are currently broadly accepted. Refactoring, in particular, was popularised by Fowler and Beck's book as

> "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [50, p. xvi].

According to this definition, refactoring transforms *correct* code to improve human-related features such as readability — or sometimes code efficiency. Since then, several studies have elaborated on the smells listed in Fowler and Beck's book. Among these, for example, are the work by Mäntylä et al. [102] and, more recently, Tandon et al. [143].

However, once again, it may be worth noting how subjective perceptions may also vary depending on the appropriateness of refactoring interventions. For instance, a study by Mäntylä [101] reported a low degree of agreement among raters as to the decision to refactor code exhibiting three simple smells from Fowler and Beck's list.

In the educational context, a preliminary step to motivate students to improve their code is providing quality feedback. This has led researchers and instructors to consider various supporting tools targeted to learners, some of which are custom-made. Their intended scope may be different: to address single code quality features, e.g., [19, 77, 84]; to provide insights while the code is being developed, e.g., [149]; to cover multiple quality criteria simultaneously, e.g., [17]; to exploit simple quality metrics such as lines of code and cyclomatic complexity, e.g., [9].

A recent comprehensive review of automatic tools to provide feedback on code correctness, maintainability, readability, and documentation can be found in Messer et al. [106]. However, as a result of an investigation on students' programs exhibiting code smells, Wiese et al. [159] reported that novices found it challenging to improve their code even when feedback was provided.

2.4 Narrowing the Scope of Code Quality for CS1

A 2023 investigation [68] asking CS1 students to rank a set of solutions from best to worst, based on their spontaneous perception, indicated students mainly consider two criteria when assessing the quality of correct code: readability and efficiency. When commenting on either or both such criteria, they referred to the choices of code structure (although their efficiency assessment was often inaccurate). Due to their lack of exposure to a full software development cycle, which occurs later in their curricula, students are not generally aware of a broad range of key quality categories identified in Table 1. To introduce code quality at the CS1 level, we focus on a subset of quality characteristics for the following reasons:

- *Exposure* — students deal with small programs, usually a procedural program or a single class with a small number of methods.
- *Relevance* — not all nine categories from Table 1 are relevant to novice programmers; for example, their code is rarely read by peers, integrated into larger projects, revisited, extended or changed later.
- *Time constraints* — CS1 courses are typically content-heavy. Hence, instructors may be limited in the time they can dedicate to this topic.

The rest of this section reflects previous work at the CS1 level and the working group's discussions on scope.

2.4.1 Adjusting code quality criteria to CS1 context. In an influential study, Stegeman et al. [139, 140] sought instructors' views to develop a rubric to assess code quality at introductory level. From the analysis of teachers' interviews, they extracted four main criteria: Documentation (names and comments), Presentation (layout and formatting), Algorithms (flow and expressions) and Structure (decomposition and modularization). Note that finer-grained characterisations of some quality aspects covered by Börstler et al.'s taxonomy [24] appear in Stegeman et al.'s rubric: *layout* and *formatting* are subcategories within readability; *names*, *headers* and (inline) *comments* pertain to documentation; and *flow*, *expressions*, *decomposition*, *modularization* and (mostly) *idiom* are connected with structural code features. The latter five structural categories were also used by Keuning et al. [82] to classify quality issues.

In an attempt to provide helpful feedback, Birillo et al. [17] have recently categorised novice programmer's quality issues into five categories: (1) code style: violations of commonly accepted style guidelines; (2) code complexity: poor design and/or overly complicated solutions; (3) error-proneness: hints to potential sources of bugs, even though not revealed by testing; (4) best-practice: disregard of widely accepted recommendations; and (5) minor issues: usually related to incorrect spelling that may hinder code readability. Notably, only one category (code complexity) covers algorithmic and structural aspects.

A *refactoring* perspective has recently received some attention at the introductory level. Two doctoral theses completed in 2020 explored novice code smells: Keuning [81] presented a tool to teach refactoring; Ureel II [148] analysed correct and incorrect code with poor structure to identify some novice patterns amenable to refactoring. Additionally, a 2023 master thesis by Leyva [91] presents a refactoring tutor that supports nine code smells split into two difficulty levels for novices.

Although all the issues listed above impact readability, when introducing code quality at CS1 level we should prioritise activities that focus on code structural issues because "[f]ixing them requires a certain level of experience and knowledge" [17, p. 311].

2.4.2 What code quality for CS1 means in this paper. We focus on quality feedback for functionally correct code, as most CS1 students can make multiple submissions in their coding assignments — receiving feedback on failed tests — until their programs appear functionally correct.

We acknowledge that CS1 courses are diverse, including different cohorts, delivery, varied choice of programming language, etc. A

primary aim of this work is to support such diversity, by assuming course conditions that are relevant to as many contexts as possible. More specifically, when discussing code quality in this paper we make the following assumptions:

CS1 content — We focus on core programming concepts: variable, selection, iteration, functions/methods. We consider decomposition and modularity in code design, but we separate object-oriented principles for those instructors who cover them in some depth.

Programming language — We refer to the languages most frequently taught at CS1 level — Python, Java, and C/C++. We, therefore, restrict our scope to examples and activities on issues that are common to these widespread programming languages.

Problem Context — Knowledge of the problem context and algorithmic approach is important when coding a solution from scratch and when reading code to extend or reuse it. However, in typical CS1 programming exercises, problem-solving may be minimal and rather tends to concentrate on implementing a given design or reusing and combining simple code plans. We then focus on CS1 programs resulting from decomposing and coding simple problems such as the *rainfall* problem [136].

Finally, although quality is often seen as a property of the product (the code final version), understanding why quality evaluation is important requires basic understanding of (a) the process of problem decomposition, design and implementation, and (b) how the product (the code in itself) will be used. To become aware of the value of code quality, students should be exposed to some extent to (a) and (b).

2.4.3 How to teach code quality. When addressing the human-oriented aspects of code quality in a learning context, it is important to consider that what is perceived as good quality depends on the subject's experience. We also need to distinguish between:

- code quality as meant for the *benefit of an individual learner*: good quality code is code that the learner can understand more easily;
- code quality as meant for *cooperating in a community*: working in a community (also a learning community) implies the need to share coding 'rules', including coding conventions, to make communication between members smoother.

In CS1 we should emphasise the individual learner and explain the benefits of adopting a clean coding style by linking the taught principles to small concrete examples and by showing students simple steps to improve their code.

From this perspective, code quality instruction will involve two main types of activities:

(1) providing rules and guidelines relative to the process leading to enhanced code quality;
(2) explaining the rationale behind rules and guidelines in connection with the quality aspects addressed.

Reflecting on the code quality at the CS1 level should make students aware of the importance of code as perceived by programmers.

3 Methodology

The aim of this research is to investigate how code quality can be addressed in the CS1 classroom (introductory programming) with a focus on examples and activities that help students become aware of and improve the quality of their code.

3.1 Research Questions

Our research questions are:

RQ 1: Which code examples related to code quality have been used in the literature on code quality in CS1 education?
  RQ 1.1: Which topics related to code quality are covered by these examples?

RQ 2: Which teaching activities related to code quality have been used in the literature on code quality in CS1 education?
  RQ 2.1: Which topics related to code quality are covered by these activities?

We used the insights from a literature review (carried out according to the guidelines outlined in the next subsection) to answer the research questions above.

3.2 Literature Review

The main purpose of the literature review is to catalogue examples and activities to support the teaching and learning of code quality at the introductory level.

3.2.1 Search strategy. Keuning et al. recently published a systematic mapping study on code quality in education [85]. Since our work has more focused objectives, we used Keuning et al.'s 195 primary references as a starting point for selecting relevant contributions. Then, we used the search terms from [85] to identify any recent papers (from Jan 2023 onwards) not captured in Keuning et al.'s study. This process identified 29 *newer* papers. In addition, limited backward snowballing resulted in a set of 24 *older* papers. More specifically, the backward snowballing process explored the references found in recent papers that satisfied the inclusion criteria described in the following subsection.

3.2.2 Inclusion and exclusion criteria. As explained above, we used the papers listed in Keuning et al.'s study, and integrated several newer and older papers. Since Keuning et al.'s review already targeted code quality in education, we focused on papers dealing with examples and activities relevant to CS1. There is, however, some disagreement about where the boundaries between CS0 and CS1 or between CS1 and CS2 can be located. For this reason, to be as inclusive as possible, we initially also considered papers targeting CS0 and CS2. The inclusion or exclusion criteria can then be summarised as described in Table 2.

3.2.3 Data extraction. A high-level description was gathered for each of the 248 papers returned from the search strategy outlined above. Each paper was reviewed by one of the members of the working group. This served to track the activity of the working group as well as to carry out the first step in the process of extracting examples and activities. Table 3 details the information that was gathered about each of the papers. Particular attention was paid to identifying papers that contain examples of code defects and activities addressing code defects used in teaching and assessment.

Table 2: Inclusion (IC) and exclusion criteria (EC) for papers

ID	Description
IC1:	The paper discusses an intervention at CS0–CS2 level addressing code quality.
IC2:	The paper provides detailed characterisations or classifications of code quality aspects or issues that are relevant for CS0–CS2.
IC3:	The paper provides examples of code quality aspects or issues at CS0–CS2 level.
IC4:	The paper discusses noteworthy insights on code quality at CS0–CS2 level.
IC5:	The paper provides guidelines or rules to comply with related to code quality at at CS0–CS2 level.
EC1:	The paper focuses on large software projects outside the scope of CS0–CS2.
EC2:	The paper focuses on industry metrics or tools with no adaptation to be used by novices at CS0–CS2 level.
EC3:	The paper discusses practices that do not translate well into practices relevant for current CS0–CS2 courses (e.g., using COBOL or the "GO TO" statement).
EC4:	The paper focuses on dynamic properties of code (e.g., program correctness or performance).
EC5:	The paper focuses on idiosyncratic features of programming languages or environments, not typical of CS0–CS2.
EC6:	The paper does not deal with teaching and learning of code quality or grading in terms of quality of code artefacts.
EC7:	The paper does not go beyond opinion-based recommendations (e.g., about using a refactoring tool).

The reviewers also included comments on the content of a paper and its applicability to the research undertaken, then listed the inclusion and exclusion criteria (from Table 2) that applied to it.

Each "included" paper was subsequently reviewed by a second member of the working group to check and possibly integrate the associated information (e.g., by indicating additional inclusion criteria). Similarly, a second reviewer was involved in some cases where the former reviewer was doubtful about excluding a paper.

3.3 Extracting and Cataloguing Examples

Now we outline the process followed to identify and catalogue examples that could be used to address code quality at the CS1 level.

3.3.1 Extracting examples from the literature. Every paper that satisfied the inclusion criterion IC3 was reviewed once again by a different member of the team to extract the reported examples. In some cases, the examples were drawn from supplementary materials linked to the paper. The examples were collected into a spreadsheet containing the fields detailed in Table 4. During the extraction process, efforts were made to be consistent with the phrasing or language used by the original authors when recording the description. When the examples of code quality issues did not have a matching example representing the resolved defect, these were added by the reviewer based on insights from the paper.

Most of the reviewed papers contained a single example of a code defect, and only in a small number of cases we retrieved more than five examples from the same study. However, two of the newer papers identified in the literature review were of particular interest in that they contained categorisations of several code defects: Silva et al. [133] list 45 "misconceptions in correct code," most of

Table 3: Description of data extraction fields for papers. (*Main sheet — one row per paper*)

Field	Description
ID	Identifier of the paper source (Kxx from Keuning et al.'s list, Nxx for recent papers after that review and Oxx for older papers found by ancestry search).
Bib data	Bibliographic data.
Reviewed by	Name of reviewer responsible for data extraction.
Quality topics	Brief description of the quality topic(s) covered.
Code examples	List of examples provided in the paper, or "None" if there aren't any. All the examples are collected in a separate sheet (see Table 4).
Level	Indicator for the course level, e.g., CS1.
Teaching	Brief description of the teaching activities or strategies from the paper, or "None" if there aren't any.
Assessing	Brief description of any code quality assessment described in the paper, or "N/A" if there aren't any.
Comments	Free text for the reviewer to comment or describe further topics.
Inclusion/Exclusion Criteria	A list of the inclusion and exclusion criteria that apply to the paper.

Table 4: Description of data extraction fields for examples (*Example sheet — one row per example*)

Field	Description
EID	Identifier for the example.
From	ID of the primary study that is the source for the example.
Code	A copy of the example code.
Issue	The issue(s) that is/are addressed by the example.
Other comments	Free text for the reviewer to comment or describe further topics.
Code Fix Example	A version of the code that removes the quality issue(s).

Table 5: Exclusion Criteria for extracted examples.

ID	Description	#n
EEC1	Related to correctness (out of scope)	12
EEC2	Related to efficiency/performance or user-friendliness (out of scope)	2
EEC3	CS0 (or earlier) defect which is not expected to persist into CS1	3
EEC4	Too high level (too large or difficult) for CS1	12
EEC5	Limited to a specific language	38
EEC6	Doubtful defect from the WG's perspective	35

which were descriptive enough to allow the generation of examples; Řechtáčková et al. [123] discuss a repertory of 80 code defects, presented in their paper available online, the majority of which include examples of both the defect and fixed code. These two comprehensive collections were included in the spreadsheet as separate sheets for ease of reference.

A final set of 282 examples was created based on the sources drawn from the literature, as described above, and by including 46 additional examples that are part of a repository of laboratory assignments administered by a member of the team.[3] Then, the exclusion criteria detailed in Table 5 were used to identify examples to be included or excluded from the final catalogue. This process was initially carried out by one member of the working group, but subsequently reviewed by others. Where any disagreement emerged, the specific example was discussed until consensus was reached, with either the example being included or the specific exclusion criteria being applied was agreed upon.

As a result, 102 examples were excluded based on the criteria in Table 5. We deemed 38 examples as too language-specific (EEC5), 20 of which come from Řechtáčková et al.'s catalogue [123] as it focused on aspects of Python that did not generalize to other common CS1 languages. In a further 35 examples, the working group members considered the defects either too marginal or, in some cases, arguably not even defects (EEC6). For instance, incrementing without using the shorthand notation was considered a very minor, even debatable defect, while "using non-English identifiers" is not a defect unless working in an international environment with specified linguistic requirements for variable names.

3.3.2 Cataloguing Examples. After identifying the candidate papers, we were left with a list of 180 examples from a variety of sources, sometimes with close similarities or slight variations between the multiple sources. Hence, we consolidated the final set by combining examples from different sources that can be considered to represent the same quality defect under a single description. While doing so, the wider working group discussed the appropriateness of each combination. This step resulted in 63 quality defects (see section 5).

The categorisation of examples started with an inductive coding process based on a smaller sample that was then refined through discussion within the working group. The classification framework agreed upon after much discussion was partly inspired by the earlier taxonomy by Řechtáčková et al. [123], where each defect is characterised in terms of the *source* of the related code quality concern and an abstract *qualifier* of the type of concern. We refined Řechtáčková et al.'s **sources** as follows:

- Instead of just *variables*, we use a generalized category called *Data* to include variables (and constants), their typing, scope, and usage as parameters or return values for methods;
- Instead of treating *conditions* as a separate source, we combine them with *Expressions* since conditions of control statements are typically expressions.
- Instead of *loops* and *functions*, we include *Control/Block* as a generalised category. (The choice and definition of functions also pertain to the *Organization* category listed next.)
- Instead of *modules*, we use *Organization* to refer to the macro-level design of a program, including its functional decomposition.
- We did not consider compound data structures as a separate category since our focus is basic CS1 topics.

As a result, we arrived at the following list of sources: *Expressions*, *Data*, *Control/Block*, *Organization*, *Documentation* and *Typographic*.

To characterise the types of quality concerns, we agreed upon six **issue qualifiers**: *Unclear*, *Duplication*, *Unnecessary*, *Missing*, *Inconsistent*, and *Unsuitable*. They correspond to defect types in Řechtáčková et al.'s categorisation as follows:

- unused is (part of) *Unnecessary*, unsuited construct is *Unsuitable*, duplicate code is *Duplication*.
- poor name, poor formatting are *Unclear*.
- poor documentation could be *Missing* or *Unclear*.
- poor design could be *Unclear*, *Duplication* or *Inconsistent*.
- simplifiable could be *Unsuitable*, *Duplication* or one of the other qualifiers in the taxonomy we propose.

3.3.3 Rates of agreement when processing the examples. The coding process relative to the examples was carried out in three stages. Four team members participated in the first stage, during which a large set of 237 examples (over 282) were coded. Each reviewer focused on a subset of examples. At the end of this stage, each example was assigned either a source-issue categorisation as described above, if it was deemed worthy of inclusion in the catalogue, or an exclusion criterion according to Table 5.

Then, in the second stage, all the examples were independently coded via a deductive process by a reviewer who was not involved in the former stage, and the rates of agreement between reviewers were evaluated as follows:

- *Inclusion vs. exclusion* – Relative to the 237 examples independently processed, the inter-rater agreement as to the decision to include or exclude a given example was 84%;
- *Exclusion criteria* – Relative to the 68 examples excluded by both independent reviewers, the rate of agreement regarding the specific reasons for exclusion was 84%
- *Classification in terms of 'source'* – Relative to the 132 examples included by both the reviewers with a precisely identified *source*, the inter-rater agreement was 80%.
- *Classification in terms of 'issue'* – Relative to the 130 examples included by both the independent reviewers with a precisely assigned *issue qualifier*, the inter-rater agreement was 72%.

(The small discrepancy between the counts in the last two points is because, in a couple of cases, an 'issue' was not assigned in the first stage.)

The reasons for disagreement could be ascribed to different interpretations of the terms labelling the categories in several cases. For instance, there were initially some ambiguities about the coverage of the term *Data* for typing, variable scoping, parameter passing, etc. This led to a more accurate definition of each term used to identify a *source* or an *issue qualifier*, clarifying the aspects it is meant to represent or not to represent. An additional cause of disagreement

[3] For ease of reference, we labelled the examples from [133] with an uppercase letter (*A*–*H*) plus a number, as in the source; those from [123] are prefixed by *R*; for all the others we used an *Ex* prefix followed by a progressive number. This is also the labelling reported in the catalogue of examples in [70] – Appendix A.

Table 6: Description of data extraction fields for activities. (*Activity sheet — one row per activity*)

Field	Description
From	ID of the primary study that is the source for the activity (there might be several sources).
Approach	Teaching approach used, e.g., direct instruction, facilitated activity, or independent activity.
Graded?	Whether the activity is graded: Yes/No.
Type of tool	Type of tool that supports the activity, if applicable, e.g., static analysis tool
Activity Type	Type of activity, e.g., improving code, applying tool.

was that sometimes multiple categories may apply; for example, the borderline between *Unclear* and *Unsuitable* is blurred (an option may be unsuitable precisely because it makes things less clear!). These cases were resolved by assigning priorities, e.g., of *Unclear* over *Unsuitable*.

Finally, in the last stage, another independent reviewer coded all the examples, and a final agreement was reached after several iterations and discussions among reviewers.

3.4 Extracting and Cataloguing Activities

Drawing from the literature review outline in subsection 3.2, our approach to collect useful code quality activities at CS1 level involved two phases as described next.

3.4.1 Phase 1 – Extracting instructional activities from the literature. Two members of our research team identified the papers that included pedagogical activities used by instructors in their course. The members determined that a paper presented a meaningful activity if at least one of the following held:

- the paper discussed an activity that was used for *instruction* rather than *evaluation*;
- the paper discussed an activity that *informed* an instructor about what to teach in lecture.

The information gathered for each activity of interest was organised according to the fields listed in Table 6.

3.4.2 Phase 2 – Cataloguing instructional activities. We then proceeded to classify the pedagogical activities. We inductively decided upon several dimensions through which to categorise them. Each paper was in charge of a single researcher. Our rationale for selecting the dimensions was to include the important characteristics of an activity that may impact whether an instructor uses said activity in their own course. We describe the dimensions below.

- **Intended level:** The student cohort for that activity:
  - *Pre-CS1:* Includes CS0 and K-12 students;
  - *CS1:* CS1 as generally understood to cover introductory programming topics regardless of the programming language (e.g., Java or Python) and approach (e.g., imperative, objects-first, functional, etc.). CS1 typically assumes no prior knowledge of programming.
  - *Post-CS1:* Refers to courses in the computer science curriculum that have CS1 as the immediate or a prior prerequisite. Examples of such courses include CS2, Data Structures and Software Engineering, or even graduate courses. An activity that can be used in both CS1 and post-CS1 courses is categorised as a CS1 activity.
- **Taxonomy Tags:** If the paper targeted aspects that are consistent with the quality principles of Kirk et al.'s taxonomy [88] (refer to subsection 4.2), then the activity was labelled with the corresponding acronyms; otherwise it was labelled as *Unspecified*.
- **Approach:** The instruction approach used in the activity:
  - *Direct instruction:* active involvement of the instructor, e.g., in a lecture, and mostly passive involvement of the student.
  - *Facilitated activity:* active involvement of both the instructor and the student, e.g., hands-on closed lab activity.
  - *Independent activity:* no or minimal involvement of the instructor, and active involvement of the student, e.g., after-class assignments, use of asynchronous tools.
- **Graded:** Whether the code quality activity is graded for course credit *with specific assessment of code quality*. Notably, if an activity was graded for course credit but none of the points were apportioned for code quality, then it was *not* considered as a graded activity from our present perspective.
- **Activity Type:** We identified the following general teaching activities in the papers:
  - *Use of Tool (No Support):* when the activity uses a tool, and the student sees an unfiltered report on the quality issues found.
  - *Use of Tool (Hints):* when the tool provides hints to improve the code rather than a list of quality defects.
  - *Peer Review:* activities where students evaluate code written by other students in the course.
  - *Theory:* activities that involve covering conceptual/theoretical (this wording is from [31, 41]) aspects of code quality.
  - *Illustrative Examples:* activities where there is some explicit description of using code quality examples to inform students.
  - *Improving Code Quality:* activities where there is an explicit iterative process of code improvement.

By the end of our classification, we listed 28 activities, drawn from papers that discuss instructional interventions at the CS1 level. The results are presented in subsection 4.4.

3.5 Categorising Code Quality Aspects

To identify an appropriate taxonomy use as the basis for categorising code quality issues, we examined papers identified during the literature review process with IC2 and IC5. These papers included classifications of code quality aspects or provided guidelines related to code quality at the CS0-CS2 level. Six authors analysed the taxonomies and compared different approaches in the related work.

Most of the identified papers focused on subsets of code quality issues for a specific study. For example, Adler et al. [2] provided a list of code flow transformations that could be automatically applied to Scratch programs to simplify control flow (e.g., *If-Else to Disjunction*), and Brewer [26] outlined documentation practices for students. Many other papers identified as IC2 were similarly

focused on categorising results from a study rather than characterising style issues in general, including function naming [29], code refactoring [67], and student perceptions of style [68].

Several other categorizations, including Breuker et al. [25], Groeneveld et al. [56], Edwards et al. [45], Iddon et al. [62], and Keuning et al. [82] addressed issues that may be automatically detected using tools such as PMD or CheckStyle. While useful for automation, these taxonomies are incomplete because they lack categories for code quality issues that cannot be automatically detected. At the other end of the scale, several papers present very high-level categories that focus on program design [152], code smells for OOP systems [12, 47, 143], or industry practices [102] but fail to characterize the kinds of issues present in code authored by students in typical CS1 courses.

After reviewing taxonomies designed to capture issues in student code, Kirk et al.'s framework [88] was selected to allow a synthesis of the various terms present in different taxonomies in a teaching-focused context. Their categorisation is informed by the widely cited rubrics developed by Stegeman et al. [139] and takes account of previous code-style studies.

After reviewing the papers identified in the literature review as IC2 and IC5, we discovered very few papers focused on a code quality taxonomy rather than presenting a classification as a by-product of work focused elsewhere. To ensure that Kirk et al.'s taxonomy provided sufficient coverage of quality issues, each group member selected an alternative taxonomy identified during the literature search and mapped each of the categories present in the paper to Kirk et al.'s principles. After completing the mapping process, the six authors discussed each categorisation decision until an agreement was reached. During this process, some items were shifted from one category to another to reflect the perceived meaning of the corresponding concept more accurately. The mapping of different code style taxonomies to Kirk et al.'s is described in subsection 4.2.

In what follows, we will refer to Kirk et al.'s categories as "quality principles" and to their framework as "taxonomy of quality principles" — or simply "taxonomy" if the interpretation of the term is clear from the context.

4 Summary of Results

This section outlines the results from the literature review and provides an overview of the examples and activities extracted from it. Sections 5 and 6 present the catalogue of examples in more detail, and the representative activities.

4.1 Literature Review

After applying the inclusion and exclusion criteria from Table 2, 130 papers were selected for further processing. The outcome of the selection process is summarised in Table 7. The papers belonging to the inclusion categories tagged IC2 and IC5, used to inform the classification terms and the taxonomy of quality principles, are the subject of subsection 4.2. From the papers listed in the IC1 and IC3 rows of Table 7 we identified examples and activities, which will be introduced in the subsections 4.3 and 4.4, respectively. We first briefly review the papers providing other heterogeneous insights (inclusion criterion IC4).

4.1.1 Noteworthy general insights from the literature. The 15 papers referenced in the IC4 row of Table 7 present interesting insights on code quality at the undergraduate level. Of those 15, five were considered exclusively for fulfilling IC4 [39, 86, 117, 132, 160], hence, we briefly focus on them below. Common to all five contributions is that they are large-scale studies or replications of existing studies (or both), providing particularly trustworthy evidence.

De Ruvo et al. [39] developed a tool to investigate code style issues in code written by CS1 students. They found that half the students submitted code exhibiting such issues, e.g., using return statements in conditionals. They found many similar issues remain in the code submitted by students in their fourth year, indicating a need for the teaching of code style.

Perretta et al. [117] investigated the usage of static analysis tools to support the grading of code style in a large CS2-course. Using such tools can substantially reduce the workload of human graders and free up some of their time for giving feedback on code quality that common tools cannot provide. The study concludes with recommendations for using static analysis tools with human graders to assess code style.

Senger et al. [132] conducted a large-scale replication study on issues identified by the tool *FindBugs* and the relationship of those issues to (a) the correctness of programs and (b) the struggle of students on programming assignments. Using data from CS1–CS3, the results showed that some issues reported by *FindBugs* are inversely correlated with program correctness, but confirmed that they correlate with struggling on larger assignments. Like Perretta et al. [117] and De Ruvo et al. [39], their work confirms that existing or custom-designed tools can be very helpful in identifying some code style issues. Such tools can help human graders objectively, consistently, and reliably identify certain code-quality-related issues.

Wiese et al. [160] replicated a study on code style. A better style is supposed to make code easier to read and understand. Teaching expert-style control flow helps CS majors to understand expert-style code as easily as novice-style code. The replication, however, cannot corroborate these results for other students. These results emphasize that we need to consider the background of our students. Students' perceptions of style and quality might differ between CS majors and other student groups. This likely also affects the type of support we need to offer different student groups.

Finally, Kirk et al. [86] examined to what extent the learning outcomes of CS1-courses worldwide (mainly USA and UK) consider code quality. They "found that only about 30% mention any code quality-related topic" and recommend "adopting agreed achievement standards for code quality." These results are symptomatic of a problematic situation. If we lack learning outcomes related to code quality, there is no formal need to teach code quality. Furthermore, even when educators perceive the topic as important, they may have difficulties justifying the resources needed for teaching and assessing code quality.

4.2 Taxonomy of Quality Principles

As previously discussed in subsection 3.5, Kirk et al.'s framework [88] was selected as a taxonomy that provided broad coverage of code style topics, along with reasoning about *why* particular decisions on code style are made in early programming courses. It

Table 7: Summary of the papers selected from the literature review by inclusion criteria.

Code	Description	References	Count
IC1	Papers that discuss an intervention or evaluation at CS1/CS2 level related to code quality.	[3] [6] [8] [9] [10] [11] [13] [17] [19] [20] [28] [29] [30] [31] [32] [34] [36] [37] [40] [41] [42] [43] [49] [52] [55] [57] [58] [60] [61] [62] [63] [64] [67] [71] [73] [74] [77] [78] [79] [83] [84] [91] [92] [93] [94] [95] [96] [98] [100] [103] [104] [105] [111] [114] [116] [120] [122] [124] [126] [131] [134] [138] [141] [142] [145] [154] [155] [159] [161] [162] [163] [164] [165]	73
IC2	Papers that provide detailed descriptions or classifications of code quality issues/areas (or a key area such as code structure or documentation).	[2] [3] [17] [24] [25] [26] [29] [32] [45] [46] [47] [56] [60] [61] [62] [67] [68] [82] [88] [89] [94] [96] [97] [102] [103] [104] [107] [108] [112] [113] [123] [135] [139] [140] [143] [144] [152] [156] [158]	39
IC3	Papers that provide examples (simple programs matching CS1 content) of some issue:	[5] [9] [22] [23] [28] [30] [32] [35] [38] [40] [47] [49] [51] [52] [53] [54] [55] [57] [59] [63] [64] [65] [66] [67] [71] [74] [75] [78] [82] [83] [84] [91] [93] [95] [97] [98] [99] [107] [111] [113] [114] [118] [119] [121] [123] [125] [133] [146] [147] [149] [154] [156] [158] [159]	54
IC4	Papers that provide interesting insights on code quality at undergraduate level.	[13] [22] [26] [39] [46] [64] [86] [94] [96] [117] [132] [145] [158] [159] [160]	15
IC5	Papers that report on (context-dependent) quality rules to comply with.	[9] [12] [27] [38] [67] [71] [97] [109] [110] [119] [131]	11

Table 8: Overview of Kirk et al.'s [88] quality principles in relation to other representative taxonomies from the literature.

Quality aspect	Description	[24]	[17]	[139, 140]	[113]
Explanatory Language (EL)	The intent and meaning of code is explicit.	✓	✓	✓	✓
Clear Layout (CL)	Different elements are easy to distinguish, and the relationships between them are apparent.	✓	✓	✓	✓
Simple Constructs (SC)	Coding constructs are implemented in a way that minimises complexity for the intended reader.	✓	✓	✓	✓
Consistent Design (CD)	Elements that are similar in nature are presented and used in a similar way.	✓	✗	✓	✓
Non-redundant Content (NC)	All elements that are introduced are meaningfully used.	✓	✗	✓	✗
Appropriate Implementation (AI)	Implementation choices are suited to the problem to be solved.	✓	✗	✓	✓
Avoid Duplication* (AD)	Code duplication is avoided.	✗	✗	✓	✗
Modular Structure (MS)	Related code is grouped together, and dependencies between groups minimised.	✓	✓	✓	✓

*Originally named "Avoid repetition" in [88].

focuses on CS1, using principles established to "support educators teaching about understanding and changing code, particularly at the beginner level" [88, p. 142]. The principles, listed in Table 8, allow a teacher to refer to specific examples relevant to their teaching context while linking to broader principles that describe *why* a given approach might be relevant to a quality measure we care about. Therefore, this work proposes an approach closely aligned with the working group's goal and could be used to frame the examples and activities used in teaching code style.

In the process of mapping various taxonomies, as shown in Table 8, we explored several other works that were adjacent to code quality taxonomies, including code smells [143], code review defects [102], pedagogic code reviews [61], design qualities in relational data modelling [152], and the seminal work by Kernighan and Plauger [80]. We have included in the table a selection of taxonomies directly related to code quality, reporting the connections with the quality aspects identified in [88].

In the following, we describe for each principle the key guidelines derived from a few influential works. A finer-grain mapping of quality indicators from related aspects in the adjacent literature is summarised in Table 9.

Explanatory Language. All the concerns brought up in Kirk et al.'s guidelines can also be found at least in one of the reviewed taxonomies. Both Kernighan and Plauger [80] and Oman and Cook [113] discuss this category in depth, providing the following advice:

- Use variable names that mean something [80];
- Choose variable names that won't be confused [80];
- Do not use cryptic or confusing variable names [113];
- Choose identifier names that make the code read well [113].

Clear Layout. Stegeman et al. [140] and Börstler et al. [24] explicitly use categories that align closely with formatting and layout. Although formatting and layout have been improved with modern IDE support, the advice aligned with this principle is still relevant today. For example, Kernighan and Plauger [80] recommends:

- Make your program read from top to bottom;
- Indent to show the logical structure of a program;
- Format a program to help the reader understand it.

Table 9: Quality principles connections to the literature.

Quality indicators for *Explanatory Language*

Key	Description
EL1	Descriptive naming [113, 139, 140]
EL2	Reasonable comments [24, 113, 139, 140]
EL3	Documentation [24], header comments [140], global and intermodule commenting [113]
EL4	Correct spelling [17]
EL5	No unnamed constants [139, 140]

Quality indicators for *Clear Layout*

Key	Description
CL1	Indentation [24]
CL2	Layout [24]
CL3	Length of code line [17]
CL4	Positioning of elements within files is optimised for readability [140]
CL5	Formatting consistently highlights the intended structure [140]
CL6	Formatting makes similar parts of code clearly identifiable [140]
CL7	No more than one task should be performed per line [139]

Quality indicators for *Simple Constructs*

Key	Description
SC1	Simple, concise expressions [24, 139, 140]
SC2	Simple control flow [17, 113, 140]

Quality indicators for *Consistent Design*

Key	Description
CD1	Naming conventions [24, 68, 113, 139, 140]
CD2	Naming vocabulary [139, 140]
CD3	Variables should not be reused for different purposes [139]
CD4	Style [24, 68]

Quality indicators for *Non-redundant Content*

Key	Description
NC1	Code is not executed [24, 80]
NC2	Over commenting [139]
NC3	No nonfunctional commented code [24]
NC4	Informative and appropriate comments [139, 140]

Quality indicators for *Appropriate Implementation*

Key	Description
AI1	Don't use conditional branches as a substitute for a logical expression
AI2	Use if-else when only one of two actions is to be performed
AI3	Well-structured
AI4	Use generic data structures where possible
AI5	Use appropriate data types.
AI6	Appropriate control structures and libraries should be chosen
AI7	Appropriate data types should be used
AI8	Use of structured constructs
AI9	Reuse appropriate library functionality [140]

Quality indicators for *Avoid Duplication*

Key	Description
AD1	No repeated expressions [139, 140]
AD2	Routines used to eliminate duplication [139]

Quality indicators for *Modular Structure*

Key	Description
MS1	Modules have clearly defined responsibilities [139, 140]
MS2	[Limit] number of line/statements in the function [17]
MS3	[Limit] the number of methods in a class [17]
MS4	Routines perform a limited set of tasks [139, 140]
MS5	Module decomposition [24, 68, 113]
MS6	Well structuredness [68]
MS7	Limited use of global variables and data structures [113]

Similarly, Oman and Cook [113] are explicit in guidance about formatting and layout:

- Do not put more than one control statement per line;
- Use parentheses to improve clarity and readability;
- Use blank lines to improve readability;
- Use spacing to improve readability.

Simple Constructs. Several of Kernighan and Plauger's recommendations [80] aim at simplifying as much as possible the program structure, both at a planning level:

- Use library functions;
- Choose a data representation that makes the program simple;

as well as at a fine-grained level:

- Avoid unnecessary branches;
- Don't use conditional branches as a substitute for a logical expression;
- Make sure special cases are truly special.

Consistent Design. Consistency allows users to interpret code more easily because they can rely on the properties of the code to have a

consistent meaning. Kernighan and Plauger remark that "unless we are consistent, you will not be able to count on what our formatting is trying to tell you about the programs. Good formatting is a part of good programming" [80, p. 148]. Similarly, Stegeman et al.'s code quality rubric suggests checking code features such as [140]:

- All names in the program use a consistent vocabulary;
- Indentation, line breaks, spacing and brackets consistently highlight the intended structure;
- Positioning of elements is consistent between files and in line with platform conventions.

Non-redundant Content. Related topics were mentioned in two of the taxonomies mapped into Table 8. While some of the categories identified from taxonomies focused on code that, when removed from the program, did not impact its functionality, neither Kernighan and Plauger [80] nor Oman and Cook [113] offer advice on non-redundant executable code. Conversely, both studies address non-redundancy of comments; for example:

- Don't just echo the code with comments — make every comment count [80];
- Avoid obvious comments, make every comment count [113].

Appropriate Implementation. The importance of this general principle is well summed up by Wiese et al.: "Instructors want students to write code that is not just functional, but also uses code structures that promote readability [...]. However, teaching students to generate code using an appropriate control structure for a task is extremely difficult and time-consuming. Research has found that 90% of introductory students used inappropriate structures for a programming task, and structure problems persist into students' 4th year of a BS degree" [158, p. 321]. The choice of *appropriate* data types and control structures are included, for instance, in Stegeman et al.'s rubric [140].

Avoid Duplication. Code that is repeated takes more effort to read, and it takes more effort to change because modifications need to occur consistently in multiple places. The core driver for writing user-defined functions in CS1 is the notion that duplicate code should be reduced by refactoring. Two of the reviewed taxonomies include some quality indicators related to this principle. Stegeman et al. [140] consider not repeating expressions an indicator for code quality and state that routines should be used to eliminate duplication. This is also consistent with advice from Kernighan and Plauger [80] to "Replace repetitive expressions by calls to a common function."

Modular Structure. All the reviewed taxonomies include some quality indicators referencing modularity or structure. However, many suggestions relate to higher-level OOP concepts such as cohesion and coupling (see, e.g., [135]) that are not typically covered in introductory programming courses and are therefore considered out of scope for this work. The key advice at this level is for each method to have clearly defined responsibilities [139, 140]. This could be likened to the advice by Kernighan and Plauger: "Each module should do one thing well" [80]. Long methods may be caused by novice programmers ignoring such advice; thus, there are generic recommendations to limit the size of functions/methods [17]. Additionally, modularity is used to avoid duplication, as discussed above.

Finally, we should note that guidelines on documentation and typographic defects are often more straightforward to give in text form instead of repetitive code examples. In contrast, describing the use of constructs or appropriate implementation is facilitated with concrete and simple examples. In subsection 7.2, the categories introduced in Table 8 will be discussed with the examples examined in this study. This will also provide a clearer characterisation of the role of each principle.

4.3 Analysis of examples

The final characterisation of code quality issues arising in small programs, identified from the literature, covers 15 types of defects in the *Data* category, four pertaining to *Expressions*, 22 to *Control/Block*, 11 to *Organization*, five to *Documentation*, and seven to *Typographic*. A detailed account of this classification, illustrated by representative code examples, is presented in section 5. The full set of retrieved materials is available at [70] (Appendix A). Note, however, that the resources are not meant to be a comprehensive inventory, as novices' code may be overcomplicated in countless ways [71].

Most examples (140 out of 180) are short code snippets having a single defect. Code choices at the block or statement level account for nearly half of those examples. This is to be expected as novices are still learning in their practice how to select the appropriate construct and how to combine constructs in the simplest way. Additionally, this number reflects the wide range of such combinations when writing a few lines of code in contrast with the range of *Documentation* or *Typographic* defects which can be easily explained or captured in a short description.

4.4 Analysis of Activities

Our review started from the 73 papers tagged IC1 in Table 7, which discuss interventions on code quality, mostly at the CS1 or CS2 level.

4.4.1 What were the intended levels? Table 10 summarises the papers by their dominant intended levels: *CS1*, *Post-CS1*, or *Other/Non-specific* (refer also to section 3, point 3.4.2). We can see from the table that 54% of the considered references pertain to CS1, 27% to subsequent undergraduate courses, whereas 19% account for the remaining cases.

However, after a deeper analysis of each paper, several of them were considered scarcely relevant or inappropriate to provide insight into activities in typical CS1 contexts. We discarded the contributions pursuing more advanced objectives (*Post-CS1*), as well

Table 10: Counts of papers tagged IC1 by intended level.

Intended Level	Count	Percent
CS1	39	54%
Post-CS1	20	27%
Other/Non-specific	14	19%
Total	73	100%

Table 11: Teaching activities extracted from the literature with the intended CS1 level

Reference	Taxonomy Tags	Approach	Activity Type	Graded
[134, 141]	EL, SC, MS	Direct Instruction	Theory, Illustrative Examples	✓
[31, 41]	Unspecified	Direct Instruction	Theory, Illustrative Examples	✓
[103, 104]	EL, CL	Direct Instruction	Illustrative Examples	
[159]	CL, SC	Direct Instruction	Illustrative Examples	
[67]	SC	Facilitated Activity	Illustrative Examples, Improving Code Quality	
[58]	CL, SC, CD	Facilitated Activity	Use of Tool (No Support), Peer Review	✓
[28]	MS	Facilitated Activity	Improving Code Quality	✓
[49]	AD, MS	Facilitated Activity	Illustrative Examples	
[155]	Unspecified	Facilitated Activity	Use of Tool (No Support), Peer Review	
[142]	SC, AI, AD, MS	Facilitated Activity	Illustrative Examples	
[61]	EL, CL, MS	Facilitated Activity	Peer Review	
[145]	AD, MS	Independent Activity	Use of Tool (Hints)	
[17]	Unspecified	Independent Activity	Use of Tool (No Support)	
[79]	MS	Independent Activity	Illustrative Examples	
[32]	Unspecified	Independent Activity	Use of Tool (Hints)	
[98]	Unspecified	Independent Activity	Use of Tool (No Support)	
[114]	Unspecified	Independent Activity	Use of Tool (Hints)	
[74]	SC	Independent Activity	Use of Tool (Hints)	
[77]	EL, CL	Independent Activity	Use of Tool (Hints)	
[9]	Unspecified	Independent Activity	Use of Tool (Hints)	
[10]	MS	Independent Activity	Use of Tool (Hints)	
[93]	CL, NC, AI	Independent Activity	Use of Tool (No Support)	
[3]	EL, CL, SC, AI, MS	Independent Activity	Use of Tool (No Support)	✓
[111]	EL	Independent Activity	Use of Tool (No Support)	
[124]	EL	Independent Activity	Use of Tool (Hints)	
[20]	Unspecified	Independent Activity	Use of Tool (Hints)	
[92]	Unspecified	Independent Activity	Use of Tool (No Support)	
[57]	EL, SC, MS	Independent Activity	Use of Tool (No Support)	

as those presenting tools intended to support teachers directly, or not in connection with some instructional intervention (*Other/Non-specific*). Moreover, we dropped eight "CS1" papers: five that discuss a setup mainly functional to research objectives and the others about the quality of testing, the role of code-quality guidelines and recommendations, and a support tool for assessment. Thus, we are left with the 31 papers referenced in connection with the 28 activities listed in Table 11, which are the focus of the rest of this section.

4.4.2 How the activities were conducted? Most of the activities require some degree of autonomous work from students, as demonstrated by the dominance of *Independent Activities* (61%) and *Facilitated Activities* (25%) shown in Table 11. Moreover, almost all the *Independent Activities* involve the use of tools (16 out of 17).

Specific *Direct Instruction* approaches were found in only four activities (14%). All of these activities use code examples for illustrative purposes, out of which only two explicitly include lecturing on the theoretical aspects of code style [31, 134]. Three activities explicitly indicated that they would further employ facilitated activities involving hands-on practice for students. Out of these, one activity [134, 141] does not provide the corrected version of the illustrative examples during instruction, but instead requires the student to improve the code style based on in-class work of assessing the code quality of the illustrative example.

Table 12 shows the breakdown of activity types that we observed. A total of 18 activities (64%) provided students with a tool (such as a static code analyzer or other hint generation system) to use during programming. Far fewer activities involved providing examples, directly lecturing students, or requiring peer code reviews. This sample illustrates the variety of activities that instructors may use to target code quality at the CS1 level. However, the papers predominantly focus on usefulness, with few of them presenting an evaluation of effectiveness. In addition, the evaluation is performed in a single context, raising questions about its applicability and the robustness of the findings in other contexts. Lastly, the evaluation methods are not standardised and universally applied across papers. These shortcomings make comparing or choosing activities challenging for potential instructors interested in teaching code quality.

Table 12: Count of activity types (percentages do not sum to 100 because one activity can be tagged with multiple types).

Activity Type	Count	Percent
Use of Tool (Hints)	9	32%
Use of Tool (No Support)	9	32%
Illustrative Examples	8	29%
Peer Review	3	11%
Improving Code Quality	2	7%
Theory	2	7%

4.4.3 Were the activities graded for code quality (and how)? The vast majority of studies (82%) did not report on grading activities in terms of code quality. As shown in Table 11, only five (18%) of the listed activities were reported in their study as being graded. Two studies *required* student submissions to pass an automatic quality check before running further tests [3, 58], which we considered as grading components related to code quality since a minimum code quality standard was a prerequisite to earning credit. One study, which presented a lightweight course on code quality, discussed an exam in which students need to refactor existing code [31]. Another study evaluated students' tests after requiring them to follow test-driven development [28]. Finally, the last study discussed an approach to assess cohesion in a CS1 students' programme, but did not include a description of using the approach to grade [134].

4.4.4 Tool Usage. Most of the teaching activities applicable to CS1 involve the use of a tool. In most cases, students are expected to use a static analyser to identify code quality issues in their programs. These activities generally involve a small amount of instruction, usually to define desirable code quality properties and how to apply the tool, and students would use the tool while carrying out their coding projects. In general, the tools are used to improve the quality of student projects, but the students are not directly assessed as part of the teaching activity. There is an even split in the teaching activities that use static analysis tools concerning what kind of feedback is provided to the student. Nine activities use tools that give students a list of quality defects in their code, and nine activities use tools that primarily give students hints on how to improve their code. Two papers, [58, 155], combine the use of a static analysis tool with peer review.

Finally, it is worth observing that two CS1-level activities diverged from the pattern of using tools to identify and fix code quality issues in student programs. McMaster et al. [103] present the *UglyCode* tool, which the instructor uses to generate specific examples of poor quality code. Carlson [28] report work in which students are taught test-driven development and code refactoring through an iterative process, where they first embed print statements to test their program, then refactor their code as they modify the testing to fit the JUnit framework.

5 Catalogue of Examples

This section presents the types of quality defects introduced in 3.3.2. The main categories addressing intrinsic structural aspects of programs — namely: *Data*, *Expressions*, *Control/Block* and *Organization* — are characterised through representative *single-smell* examples in the tables 13–21. We have included one variation of each defect, except for those that are self-explanatory.
Tables 13 through 21 are organised as follows:

(i) The caption reports the *source* and *qualifier* (see 3.3.2).
(ii) For each type of defect the bibliographic sources of the examples are listed on the right of the description; if more than one appropriate example can be found in the same source, the overall number of items is shown between round brackets.
(iii) On the left side of each example, the specific bibliographic reference (from the list mentioned above) is reported once again.

For the *Documentation* and *Typographic* categories, and for defects concerning more advanced programming topics, we have provided only a general description in Table 22 and Table 23, respectively. A more comprehensive set of defects is available online at [70] under Appendix A.

5.1 Refactoring at Novice Level

Since most of the examples involve changes in the code structure (i.e., refactoring), we describe the implications of interventions appropriate for an introductory programming context.

As pointed out by Fowler and Beck [50], a code smell is a surface indication of a deeper problem. They made two subtle points about a code smell:

(1) you need to **spot** it: "a smell is by definition something that's quick to spot … a long method is a good example of this — just looking at the code and my nose twitches if I see more than a dozen lines of Java."
(2) you have to **decide** if a smell actually needs cleaning: "smells don't always indicate a problem. Some long methods are just fine. You have to look deeper to see if there is an underlying problem there."

Making a decision as to whether or not to refactor is an additional challenge for novices. Therefore, we have favoured code smells that always need fixing: all but four of the code smells addressed in the catalogue can always be refactored by applying a simple-to-describe transformation. In the following tables, the four smells that may or may not be refactored have been marked with a star (*). By including this small set in the CS1 catalogue, CS1 instructors have the opportunity to explain that some other patterns, such as CS2 smells, require programmers to make a judgement on whether they will improve readability and simplicity when cleaned.

5.2 Quality Defects – Data

We have identified a total of 15 data-related defects in the literature. Twelve of them are described with examples in the tables 13–15. The remaining three defects involve OOP concepts and will be discussed later.

The first four defects relate to different redundant uses of variables. Students commonly declare a variable that is either never used in later statements (DT1), or is only used once and could be removed (DT4*). The second case (DT4*) may not be considered a defect if a temporary variable adds clarity to the code. A method may also include unused data: it may be an unused parameter (DT2), or a return value that is not needed by the caller (DT3). Note that

Table 13: Quality defect descriptions for Data – Part 1: Unnecessary

DT1. Unused or dispensable variable [40], [133], [123]

[40]

```
int main(){
    int a;
    a = 5;
    // other code
    return 0;
}
```

```
int main(){
    // other code
    return 0;
}
```

DT2. Unused or dispensable parameter [123], [90] *(2)*

[90]

```
int inputHour( int hour ) {

    do {
        cout << "Enter the hour (1-12)" << endl;
        cin >> hour;
    } while( !(hour >= 1 && hour <= 12) );

    return hour;
}
```

```
int inputHour() {
   int hour;
   do {
      cout << "Enter the hour (1-12)" << endl;
      cin >> hour;
   } while( !(hour >= 1 && hour <= 12) );

   return hour;
}
```

DT3. Unused return value [90] *(2)*

[90]

```
int swap( int & first, int & second ) {
    int temp;
    temp = first;
    first = second;
    second = temp;
    return temp;
}
```

```
void swap( int & first, int & second ) {
    int temp;
    temp = first;
    first = second;
    second = temp;
}
```

DT4*. Variable value used only once [71]

[71]

```
long newRow(int N, vector<long> comb){
  long res = 0;
  if (N <= 0){
    res = accumulate(comb.begin(),comb.end(),0l);
    return res;
  }
  vector<long> copy = comb;
  for (int i = 0; i < comb.size(); i++){
    if (comb.at(i) != 0){
      comb.at(i) = copy.at(i-5) + copy.at(i+5) +
                   copy.at(i+1) + copy.at(i-1);
    }
  }
  res = newRow(N-1,comb);
  return res;
}
```

```
long newRow(int N, vector<long> comb){

  if (N <= 0){
    return accumulate(comb.begin(),comb.end(),0l);
  }
  vector<long> copy = comb;
  for (int i = 0; i < comb.size(); i++){
    if (comb.at(i) != 0){
      comb.at(i) = copy.at(i-5) + copy.at(i+5) +
                   copy.at(i+1) + copy.at(i-1);
    }
  }
  return newRow(N-1,comb);
}
```

Table 14: Quality defect descriptions for Data – Part 2: Duplication (DT5–6), Unclear (DT7–9)

DT5. Two variables with same role / same values [75] *(2)*, [71]

[71]

```
values = []
between = m-1

for i in range(m,n+1):
    between += 1
    if between%6 != 0:
        if between%3 == 0:
            values += [between]
        elif between%2 == 0:
            values += [between]
print(values)
```

```
values = []

for i in range(m,n+1):
    if i % 6 != 0:
        if i % 3 == 0:
            values += [i]
        elif i % 2 == 0:
            values += [i]
print(values)
```

DT6*. Excessive use of variables or data structures to store intermediate values [133] *(2)*, [90]

[133]

```
condAux = cond1 and cond2
condRes = condAux and cond3
```

```
condRes = cond1 and cond2 and cond3
```

DT7. Use of integers 1/0 with *Boolean* role [133]

[133]

```
if var1 > 0:
    cond1 = 1
else:
    cond1 = 0
if var2 > 0:
    cond2 = 1
else:
    cond2 = 0
if cond1 + cond2 == 2:
    func()
```

```
cond1 = var1 > 0

cond2 = var2 > 0

if cond1 and cond2:
    func()
```

DT8. Global Variable when only local scope needed [98], [149], [123]

[98]

```
global_list = []
def main():
    global_list.append("abc")
    print(global_list)
    return None
main()
```

```
def main():
    local_list = []
    local_list.append("abc")
    print(local_list)
    return None
main()
```

DT9. Pass-by-reference parameter instead of pass-by-value [75], [90] *(2)*

[90]

```
void printTime( int &hour, int &min, int &sec ) {
    cout << hour << "":"" << min << "":""  << sec;
}
```

```
void printTime( int hour, int min, int sec ) {
    cout << hour << "":""  << min << "":""  << sec;
}
```

Table 15: Quality defect descriptions for Data – Part 3: Inconsistent (DT10–11), Unsuitable (DT12)

DT10. Inconsistent typing of variables [90]

[90]

```
// Variable declarations
float checkAmount;
long double tipPercent;
double tip;
```

```
// Variable declarations
float checkAmount;
float tipPercent;
float tip;
```

DT11. Mixed use of parameter-passing and return value [90]

[90]

```
int inputTime( string occasion,
               int &hour, int &min, int sec ) {

    hour = inputHour( occasion );
    min = inputMinute( occasion );
    sec = inputSecond( occasion );
    return sec;
}
```

```
void inputTime( string occasion,
                int &hour, int &min, int &sec ) {

    hour = inputHour( occasion );
    min = inputMinute( occasion );
    sec = inputSecond( occasion );

}
```

DT12. Pass by reference instead of return value [90]

[90]

```
void convertToCelsius( double fahrenheit,
                       double & celsius ) {
    celsius = (fahrenheit - 32) * 5.0 / 9.0;
}
```

```
double convertToCelsius( double fahrenheit ) {
    return (fahrenheit - 32) * 5.0 / 9.0;
}
```

Table 16: Quality Defect descriptions for Expressions

Unnecessary

EX1. Redundant Boolean comparison [67], [97], [121], [133], [123]

[121]

```
if ((x > 10) == true)
  return length;
```

```
if (x > 10)
   return length;
```

EX2. Redundant arithmetic operations or typecasts [133], [123]

[133]

```
var1 = str(input())
var2 = int(6 + 4)
```

```
var1 = input()
var2 = 6 + 4
```

Duplication

EX3. Repeated expression whose value is already known [123]

[123]

```
if len(lst1) == 0:
    print(lst2[len(lst1)])
```

```
if len(lst1) == 0:
    print(lst2[0])
```

Unclear

EX4. Simplifiable expression based on Boolean logic and relational operators [123]

[123]

```
not a <= b
```

```
a > b
```

parameters could be viewed as pertaining to the Data category (as we have done in this catalogue) or as part of modular coding, which would fit into *Organization*.

Variables may be duplicated, playing the same role but with different names. Table 14 shows one such example (DT5), where the variable *between* holds always the same value as the loop control variable. Incidentally, defects of this type highlight the need for feedback on variable roles. Other examples of duplication involve the use of unnecessary variables to store intermediate results (DT6), or using integers instead of Booleans (DT7).

Unclear usage of variables may relate to their scope. For example, a variable is declared with global scope but is only used locally (DT8), or a parameter is passed by reference although only its value is needed (DT9). Students also make inconsistent or unsuitable choices for (variable) data types, return commands, or parameter-passing as shown in Table 15 (DT10, DT11 and DT12).

5.3 Quality Defects – Expressions

We found four defects in the literature at the expression level. They are illustrated by the examples in Table 16. The most cited and frequently found is the unnecessary check of a Boolean expression (EX1). A typecast to the current type is also a redundant operation (EX2). Moreover, we often find duplicated expressions (see also CB11), although when an expression has been evaluated there is typically no need to re-evaluate it (EX3).

Finally, there are multiple ways to write the same logical expression, but we should prefer the simplest one (EX4). We note that Wellesley College provided a great public online resource that explains Boolean logic simplification for novices [33]. Students may not write complex expressions from scratch, but they may be guided to assemble them when applying particular pattern-based refactoring rules, such as those presented later (see, e.g., CB8).

5.4 Quality Defects – Control/Block

As students learn to use and combine basic programming constructs, it is not surprising that this turns out to be the main source of code smells at the introductory level. Indeed, a range of verbose patterns, either unnecessary or duplicated, has been recurrently documented in the literature. Tables 17 to 19 provide a list of such patterns, starting with unnecessary statements, then covering duplication, and finally other unclear or unsuitable code structures.

Redundant statements. The explicit comparison of Boolean values, as in EX1, also translates into a verbose check in order to return True/False (CB1), when it would be simpler to just return the value of the Boolean expression directly. Checking conditions when it is not necessary, in connection with cascaded *if* statements (CB2), is another example of redundancy. Similarly, the value of some conditions in a complex Boolean expression may be already known for a given execution flow, hence such conditions can be removed (CB3). Depending on the program structure, similar patterns can sometimes be easy to spot, but other times it may be a challenge for a novice to detect a redundant or ineffective (CB4) statement. For example, it is easy to see that any statement immediately after a return is not reachable (CB7), or that an empty or ineffective conditional branch (CB5) can be removed. As an additional instance of redundancy, nested *if* statements can be combined if they both have the same (possibly empty) else block (CB8).

Duplication. Duplication is quite common in students' code, ranging from replicated computations that could be saved (CB9*) to mergeable subsequent conditionals with either the same body (CB10) or the same condition (CB11). A very frequent case is to have the same calculation performed in all the alternative branches of a conditional or switch construct (CB12). By factoring that statement out of the construct (before or after, according to data dependencies) it is clear it always executes, regardless of the condition's outcome.

Inappropriate choice of flow-control constructs. Common defects affecting novice code can be attributed to inappropriate choices of a control construct, such as nesting instead of cascading conditionals (CB21), or using a *while* instead of a *for* loop (CB20), or vice versa. Modifying the control variable inside the body of a *for* loop (CB15) may be an indication that using a *while* or *do-while* would be more appropriate.

Poor treatment of iteration boundary cases. More subtle defects in connection with loops depend on the way the boundary cases are dealt with and require a thoughtful restructuring of the code, possibly moving parts inside or outside the loop body. A *while* condition tested again in the middle of its block (CB13) or some special treatment of a boundary case within the loop (CB16) may be clues of related smells. Including a statement within a loop that is only executed in the last iteration (CB17) and infinite loops with breaks (CB22) are other possible examples of this type of defect.

Other defects at the flow control level. Finally, the list of defects in the *Control/Block* category covers multiple *for* loops with the same iteration range that could be merged into a single loop (CB14*; example not shown in Table 18), as well as two less frequent patterns: A single-iteration *while* loop used instead of an *if* conditional (CB18) and the use of recursion when an iteration would be a simpler solution (CB19).

5.5 Quality Defects — Organization

Organization is critical to code quality, with most code smells addressed by professional developers being at this level. However, since CS1 students write small programs, we found only eight defects related to this category, listed in Tables 20 and 21.

Students may write a function that is never used (OR1), or add specific code to deal with boundary cases that are already captured in a loop's general case (OR2). Sometimes, they disregard code patterns for which the computation could be extracted into a method (OR3), or they may define multiple functions that perform almost identical tasks (OR4). In the last two cases, students fail to generalise for better code reuse, whereas in other cases, the functions may be casually spread over the code (OR5).

Similarly, the order of parameters in the method signatures may be inconsistent (OR7) — a simple code smell that is easy to spot and fix — or the *Single Responsibility Principle* may be ignored when decomposing the code into multiple methods (OR6; there are many varied examples, but only two are shown). We note that a long method is often an indicator of poor decomposition (OR8*).

Table 17: Quality defect descriptions for Control/Block – Part 1: Unnecessary

CB1. Redundant if statement that could be replaced with return [52], [67] *(2)*, [123]

[52]

```
if (num % 7 == 3)
    return true;
else
    return false;
```

```
return num % 7 == 3;
```

CB2. Redundant (or consecutive) if condition that could be replaced with else [67] *(2)*, [91] [133], [123]

[123]

```
if x < y:
    print(1)
elif x >= y:
    print(2)
```

```
if x < y:
    print(1)
else:
    print(2)
```

CB3. Redundant check of a condition (part of a composite condition) [65], [133]

CB4. Statement with no effect (or made ineffective by a subsequent statement) [75], [83], [133] *(2)*, [123]

[133]

```
var = ''
var = int(input())
```

```
var = int(input())
```

CB5. Ineffective conditional branch [123]

[123]

```
if c:
    pass
else:
    # body
```

```
if not c:
    # body
```

CB6. Useless loop - only a single iteration needed [149], [133], [123]

[149]

```
public double abs(double d) {
    double result = d;
    for (int i = 0; i < 10; i++) {
        if (d < 0) {
            result = -d;
        }
    }
    return result;
}
```

```
public double abs(double d) {
    if (d < 0) {
        return -d;
    }
    return d;
}
```

CB7. Unreachable code [133], [123]

[123]

```
return x
print(x)
```

```
return x
```

CB8. Nested if statements that can be collapsed into one using AND [67], [91], [123]

[123]

```
if c1:
    if c2:
        # body
```

```
if c1 and c2:
    # body
```

Table 18: Quality defect descriptions for Control/Block – Part 2: Duplication

CB9*. Saving repeated computation into temporary value [67], [123]

[67]

```
if (value > 10) {
    p = p + ((c + 10) * TAX_RATE) / 100;
} else {
    p = p - ((c + 10) * TAX_RATE) / 100;
}
```

```
double adj = ((c + 10) * TAX_RATE) / 100;
if (value > 10) {
    p = p + adj;
} else {
    p = p - adj;
}
```

CB10. Consecutive if statements with *same body* could be combined (OR) [67], [133], [123]

[67]

```
if (value < 2) {
    return 0;
}
if (value > 15) {
    return 0;
}
```

```
if ((value < 2) || (value > 15)) {
    return 0;
}
```

CB11. Consecutive if statements with *same condition* that can be merged [133], [90]

[133]

```
if num1 == num2 * 2:
    print(num1, "multiple of", num2)
if num1 == num2 * 2:
    print(num1, "even")
```

```
if num1 == num2 * 2:
    print(num1, "multiple of", num2)
    print(num1, "even")
```

CB12. *Same code* within all conditional branches can be "factorised" [52], [67], [69], [75] *(2)*, [91], [123], [90]

[123]

```
if cond:
    print("foo")
    i += 1
else:
    print("bar")
    i += 1
```

```
if cond:
    print("foo")
else:
    print("bar")
i += 1
```

CB13. While condition tested again in the middle of its block [133], [90]

[133]

```
varCond = # setting initial value
while varCond != 0:
    varCond = (...) # getting new value
    if varCond == 0:
        break
    # code performing some function
```

```
varCond = (...)
while varCond != 0:
    # code performing some function
    varCond = (...) # getting new value
```

CB14*. Multiple distinct loops that operate over the same data [133]

Table 19: Quality defect descriptions for Control/Block – Part 3: Unclear (CB15–17), Unsuitable (CB18–22)

CB15. Modifying *for* control variable inside loop [133], [123]

[133]

```
for iter in range(numMaxIter):
    iter = ( ... )
```

```
for iter in range(numMaxIter):
    other_var = ( ... )
```

CB16. Special treatment of boundary case(s) inside *for loop* body [66], [133]

[66]

```
System.out.print("{");
for(int i = 0; i < testArray.length; i++){
     if (i == testArray.length - 1) {
        System.out.print(testArray[i] + "]");
     } else {
        System.out.print(testArray[i] + ", ");
     }
 }
```

```
System.out.print("{");
for(int i = 0; i < testArray.length-1; i++){
     System.out.print(testArray[i] + ", ");

}
System.out.print(testArray[i] + "]");
```

CB17. Statement inside loop only needed in last iteration [133], [90]

[133]

```
for iter in lst:
    var1 += 1
    var2 += iter
    var3 = var2 / var1
```

```
for iter in lst:
    var1 += 1
    var2 += iter
var3 = var2 / var1
```

CB18. Single-iteration *while* loop used instead of *if* [75], [133]

[133]

```
while num1 + num2 < 9:
    print(num1 + num2, "has more than 1 digit.")
    break
```

```
if num1 + num2 < 9:
    print(num1 + num2, "has more than 1 digit.")
```

CB19. Use of recursion instead of a simple iteration [90]

CB20. Inappropriate choice of a loop construct (e.g., *while* instead of *for*) [114], [133], [123] *(2)*, [90]

[90]

```
// Print all the even numbers less than limit
number = 0;
while( number < limit ) {
    cout << number << endl;
    number = number + 2;
}
```

```
// Print all the even numbers less than limit
for( n = 0; n < limit; n = n + 2 ) {
    cout << n << endl;
}
```

CB21. *if* nested inside else branch when *elif/else-if* is possible [123]

CB22. Infinite loops with breaks [90] *(2)*

[90]

```
do {
    cout << "Enter an even number" << endl;
    cin >> number;

    if ( number % 2 == 0 )
        break;
} while( true );
```

```
do {
    cout << "Enter an even number" << endl;
    cin >> number;
} while( number % 2 != 0 );
```

Table 20: Quality defect descriptions for Organization – Part 1: Unnecessary (OR1–2), Duplication (OR3–4), Unclear (OR5)

OR1. Unused function (or similar unit)

[123]

```
def f():
    print(4)

print(4)
```

[123]

```
print(4)
```

OR2. Special case is already captured in general case

[123]

```
if len(lst) == 0:
    return 0
sum = 0
for i in range(len(lst)):
    # body
return sum
```

[159], [123]

```
sum = 0
for i in range(len(lst)):
    # body
return sum
```

OR3. A code block that appears multiple times

[123]

```
if c:
    f(1)
    g(1)
    f(1)
else:
    f(2)
    g(2)
    f(2)
```

[123]

```
def fun(n):
    f(n)
    g(n)
    f(n)

if c:
    fun(1)
else:
    fun(2)
```

OR4. User-defined functions doing essentially the same work

[90]

```
void printLatitude( string city,
                    int deg, int min, int sec ) {
  cout << "The latitude of " << city  << " is "
       << deg << ":" << min << ":" << sec << endl;
}

void printLongitude( string city,
                     int deg, int min, int sec ) {
  cout << "The longitude of " << city  << " is "
       << deg << ":" << min << ":" << sec << endl;
}
```

[49], [90]

```
void printL( string city, string latLongItude,
             int deg, int min, int sec ) {
  cout << "The " << latLongItude << " of "
       << city << " is "
       << deg << ":" << min << ":" << sec << endl;
}
```

OR5. Arbitrary organisation of declarations

[133]

```
var1 = input ()
def func1():
    (...)
var2 = input ()

var3 = func1(var1, var2)
def func2():
    (...)
var4= func2(var3, var1)
var5 = input()
```

[133], [90]

```
def func1():
    (...)
def func2():
    (...)

var1 = input ()
var2 = input ()
var3 = func1(var1, var2)
var4=  func2(var3, var1)
var5 = input()
```

Table 21: Quality defect descriptions for Organization – Part 2: Unclear (OR6), Inconsistent (OR7), Unsuitable (OR8)

OR6. Lack of decomposition [22], [28], [91], [90]

[90]

```
string inputStatus() {
    // Input the number of credits taken
    int credits;
    do {
        cout << "Enter the number of credits taken"
             << endl;
        cin >> credits;
    } while( !(credits >= 1 && credits <= 150) );

    string status;
    if( credits <= 28 )
        status = "freshman";
    else if( credits <= 56 )
        status = "sophomore";
    else if( credits <= 84 )
        status = "junior";
    else
        status = "senior";
    return status;
}
```

```
int inputCredits() {
    int credits;
    do{
        cout << "Enter the number of credits taken"
             << endl;
        cin >> credits;
    } while( !(credits >= 1 && credits <= 150) );
    return credits;
}

string inputStatus( int credits ) {
    string status;
    if( credits <= 28 )
        status = "freshman";
    else if( credits <= 56 )
        status = "sophomore";
    else if( credits <= 84 )
        status = "junior";
    else
        status = "senior";
    return status;
}
```

[91]

```
def euclidean_distance(coord1, coord2):
    '''
    coord1: (x1, y1), coord2: (x2, y2)
    '''
    return (abs(coord1[0] - coord2[0]) * abs(coord1[
    ↪  0] - coord2[0]) + abs(coord1[1] - coord2[1])
    ↪  * abs(coord1[1] - coord2[1])) ** (1 / 2)
```

```
def euclidean_distance(coord1, coord2):
    '''
    coord1: (x1, y1), coord2: (x2, y2)
    '''
    x_dist = abs(coord1[0] - coord2[0])
    y_dist = abs(coord1[1] - coord2[1])
    return (x_dist*x_dist + y_dist*y_dist) ** (1/2)
```

OR7. Inconsistent order of (the same) parameters in different methods [90]

[90]

```
void printDate( int day, int month, int year ) {
    cout << month << "-" << day << "-" << year
         << endl;
}

void inputDate( int &day, int &year, int &month ) {
    year = inputYear();
    month = inputMonth();
    day = inputDay( year, month );
}
```

```
void printDate( int year, int month, int day ) {
    cout << month << "-" << day << "-" << year
         << endl;
}

void inputDate( int &year, int &month, int &day ) {
    year = inputYear();
    month = inputMonth();
    day = inputDay( year, month );
}
```

OR8*. Long function/method/script [59], [123] *(2)*

[123]

```
def f():
    ... # > 20 statements
```

Decompose further if possible

Table 22: List of quality defect descriptions for Documentation and Typographic issues.

Documentation	Unnecessary	DC1. Useless or ineffective comment	[123]
	Unclear	DC2. Variables and functions with meaningless or misleading names	[75], [91], [133], [123] *(2)*
		DC3. Using magic numbers in expressions (including ASCII codes)	[91] ,[133], [123]
	Missing	DC4. Missing header comment	[133], [123]
		DC5. Missing blank lines	[63]
Typographic	Unclear	TP1. Too many items on a single line / line of code too long	[133], [123] *(2)*
		TP2. Messy layout and formatting	[63]
		TP3. Long inline comment	[133]
	Missing	TP4. No indentation, no blank lines separating blocks	[133]
	Inconsistent	TP5. Inconsistent indentation, spacing and naming (syntactic) conventions	[113], [30], [91], [123] *(3)*
	Unsuitable	TP6. Unsuitably composed identifiers (do not adhere to established conventions)	[123]

Table 23: Examples descriptions by (source, issue qualifier) – for less pervasive CS1-related topics.

Data	Unclear	DT13. Method/function parameter list can be made more compact	[123], [90]
	Unnecessary	D14. Instance variable instead of method local variable (OOP)	[90]
	Unsuitable	D15. Public instead of private instance variable (OOP)	[121]
Organization	Unclear	OR9. Excessive use of overloading (different function/method signatures)	[52]
		OR10. Distribution of functionality among "casual" classes (OOP issue)	[95]
	Unnecessary	OR11. Re-implementing library code	[69]

5.6 Quality Defects – Documentation and Typographic

Although documentation, layout and formatting are important for readability, only a limited number of examples from the literature help to illustrate the related quality defects. We identified five generic documentation defects and seven typographic defects which are listed in Table 22. Most such cases are able to be detected automatically, and the solution is straightforward but context-dependent (e.g., missing documentation can be included, but there is no generalisable pattern that can be used to model the solution). It is rarely the case that a program has only one documentation or typographic defect, so we have grouped multiple minor issues into common descriptors, e.g., TP4 or TP

5.7 CS1 Advanced Examples

As mentioned previously, some examples address knowledge that is not consistently covered in all CS1 courses but may be useful for some CS1 designs. Table 23 shows the six patterns that fall into that broad category. Four of them are related to OOP issues, one to the use of compact data structures (DT13), and the last pattern, re-implementing library code (OR11), is often due to a lack of familiarity with the standard libraries.

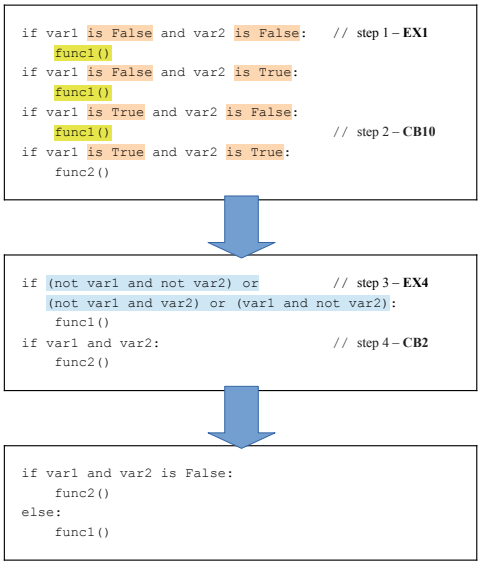
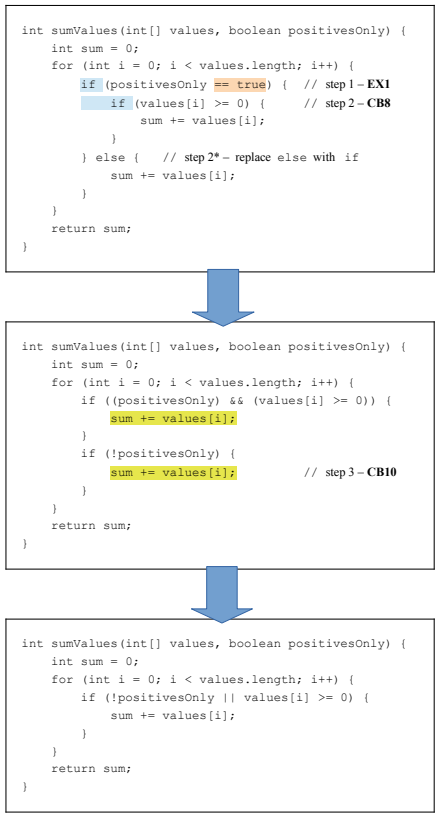


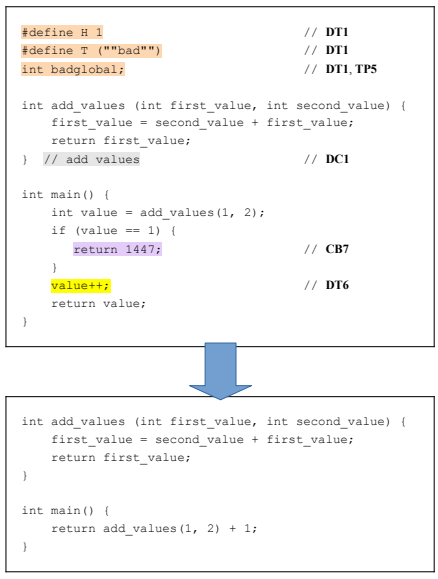
Figure 1: An heterogeneous example E1 from [133] and the steps to clean the code

5.8 Refactoring Code with Multiple Defects

The examples described in Tables 13 to 21 contain only one main defect, so that students can learn how to edit or refactor each issue individually. However, it is not uncommon for student code to contain multiple defects as indicated by 21 examples in our dataset. (The rest of the filtered examples, though appropriate for the CS1 level, required a more substantial reworking of the code structure, that cannot be achieved by applying specific refactoring rules.) In the cases of multiple defects it is advisable to identify and refactor one issue at a time. We will describe this process with three different examples, shown in Figure 1 and Figure 2.



(a) Refactoring example Ex4 from [84]



(b) Refactoring example Ex121 from [57]

Figure 2: Refactoring two additional heterogeneous examples

Code pattern E1 from Silva et al.'s catalogue [133] is described as having one issue: "Checking all possible combinations unnecessarily". However, as shown in Figure 1 we can identify two related issues: the unnecessary evaluation of each Boolean variable (EX1) and the fact three of the listed conditions have the same body. These can be combined (CB10) to obtain a complex Boolean expression that can be simplified. After the condition is reduced to "*not var1 or not var2*", we can see the two *if* conditions are complementary and hence we can reduce them to an *if/else*. Although we may be able to go directly from the first version to the last, it is likely many novice programmers will require the intermediate steps and, in some cases, they may not detect all quality defects, producing only a partial refactoring.

The second example, from [84], is shown in Figure 2(a), and has been used in multiple refactoring studies. After removing the Boolean comparison (EX1), we need to make minor adjustments to the code in order to compensate for the empty *else* statement of the nested *if*. This requires an intermediate step to separate the *else* (reversing a CB2 transformation) so that we can first combine the nested *if*s and then merge the resulting sequenced *if* statements with "OR rule" (CB10).

The third example from [57] has multiple defects pointing to a range of sources as labelled in Figure 2(b) The typographic and redundant variable definitions (DT1) can be fixed in any order as they are independent. Note that the *if* condition is always false, hence the first *return* is unreachable code (CB7). Once we remove that statement, we could also remove the local variable that stores the result from *add* and use the value of the expression directly (DT6). One problem with this example, however, is that it is scarcely interesting since the code itself is doing very little.

The examples above illustrate that the order in which we can deal with multiple defects depends upon the potential code dependencies. During the transformation process, it is also possible that new code smells are created, which must be fixed in turn, as can be seen in Figure 1.

We should note that poorly structured code with a large number of interconnected defects could be quite hard to disentangle. In such cases, when we are unable to fix one issue at a time, it may be easier to rewrite the block from scratch instead.

6 Sample Activities

In this section, we describe a selection of the most representative types of activities identified in our literature review (Table 11) in more detail. For each activity, we focus on questions of interest by the authors in their roles as educators, such as: how well is the protocol of the activity described? What is the extent of the materials provided? What is the setup cost? Is the activity targeted to specific topics? Does it scale well?

Proposal: Activity Cards. A recurring theme that emerged during the working group discussions is the difficulty in identifying an activity that a teacher may adopt for their own context, based on the literature in which an example is introduced. We understand this to be a constraint of efforts to publish works in a scientific format; sacrifices are made in terms of where the focus is, as there simply is not enough space for a deep account of how to run the activity, what tools were used, and what arrangements were required.

Our contribution is to present a collection of *Activity Cards* that are easy to understand, whilst conveying enough salient information for the teacher to help support their instruction. We have

prepared nine activity cards as exemplars — a selection from the 28 activities discovered in the literature. Five of them are included in this section for reference; the others can be found in [70] under appendix B.

At a more general level, in subsection 7.4 we will discuss a couple of recommendations concerning the introduction of code quality-related activities in typical CS1 contexts.

6.1 Direct Instruction Activity

McMaster et al. [103, 104] proposed a tool for generating examples with limited documentation and poor typographic style to be used in lecture (see also the "UglyCode" card). The setup cost hinges on the availability of the tool, which the authors mention can be obtained upon request via email. Despite the lack of a formal evaluation of its effectiveness, this activity is relatively scalable. Although the idea can be expanded to different languages and features, the tool was implemented in C++.

UglyCode [103, 104]

Description: The instructor can create examples for students that add extra blank lines or remove them, add random indentation lengths or remove indentation, adds garbage comments, change variable names, increase line lengths. Instructor can pair this with tools that improve code quality to demonstrate that good style leads to easier to understand and maintain code.

Quality Focus: Descriptive variable naming (EL), reasonable comments (EL) indentation and layout (CL), line length (CL).

Materials or Tools: A tool called UglyCode that will take Java or C++ code and add code smells to it.

Grading or Assessment: None

Supported Language(s): Java or C++

6.2 Facilitated Activities

In a recent paper, Tan and Poskitt [142] describe an activity ("Fix Your Own Code Smells" card) that tries to emulate the real-world scenario of a refactoring task by requiring students to complete a programming exercise designed in such a way that ensures they will produce a code smell. Then, they are guided to refactor their own code and appreciate the improvements attained in terms of code cleanliness.

Fix Your Own Code Smells [142]

Description: This activity applies a mistake-based approach to learning about code quality. First, students are provided a programming exercise with instructions that result in a code smell being introduced. Then after completing the exercise, students learn about the relevant code smell(s) that were introduced, then asked to identify the smells in their own code and then modify their code to fix the smells.

Quality Focus: Long Method, Long Parameter List (SC); Duplicate Code (AD), Data Clumps Large Class, Primitive Obsession (SC).

Materials or Tools: A full set of exercises provided here.

Grading or Assessment: None

Supported Language(s): Python

Similarly, Izu et al. [67] present a facilitated activity in which students are provided with a lightweight resource of refactoring examples to use during a lab session. The resource consists of four rules that can be applied to simplify conditional constructs ("Refactoring Conditional Statements" card).

Refactoring Conditional Statements [67]

Description: Students are given a resource about four code quality "rules" related to conditional statements and tested on their ability to apply those rules and write shorter/simpler code. The authors also found a persistent effect of their intervention two weeks later, with students who received the handout with code quality rules writing shorter code than students who did not receive the handout.

Quality Focus: Simplifying IF/ELSE statements (SC), Removing duplicate (AD) or redundant statements/logic in IF/ELSE statements (NC).

Materials or Tools: This resource includes all the materials given to students: https://iticse22-refactor.github.io/resource/.

Grading or Assessment: The grading for this activity was only focused on functional correctness. Since the activity required refactoring correct, but low quality, code, the students could get a full grade by not making any modifications.

Supported Language(s): C

6.3 Independent Activities

A large proportion of activities that promote code quality identified in Table 11 are tool-based independent activities that were (partially) motivated by addressing the issue of scale. As a result, tool use becomes the activity, which is problematic in terms of ensuring students are engaging in (and learning from) the activity. However, given the resource challenges in larger courses, these efforts are worthy of attention to give a sense of what could be included as a mostly passive code quality activity that scales well.

The goal of Choudhury et al.'s work [32] is to provide a scalable solution to generating formative code style feedback for students working on an assignment. The tool presented, *AutoStyle* (see related card), is unfortunately not available in any way that would make it immediately usable to instructors — a common issue with novel tools within computer science education and more fully explored in a previous working group [18].

Nevertheless, the approach is worthy of detailing here. The two most interesting aspects of this work are that: (1) it reduces the effort of instructor-guided feedback from being proportional to all students to clusters of similar submissions; (2) it complements instructor-guided feedback with automatically generated feedback.

Previous student submissions or instructor-generated solutions can be used to seed the process. From this dataset, clusters are identified based on their abstract syntax tree edit distance. Then these clusters are manually tagged by instructors as being good, average or weak, as well as labelled with a guidance note on how to improve the code style. To complement this, near-but-better solutions can be stripped back to a code skeleton in order to help a weaker student move forward in a progressive manner (rather than leap to the best solution immediately). Syntax hints are also generated, e.g., reporting which built-in functions or structures could be used.

AutoStyle [32]

Description: A novel feature of this approach is that previous student submissions are used to identify poor code quality. These clusters are labelled good, average or weak. This has the effect of reducing the scale, as the scaling factor is no longer the number of student submissions, but the number of clusters discovered. This reduces the effort in generating feedback and exemplars.

Quality Focus: Assignments, branches, and conditional statements are counted to form an ABC score. The main aim is to simplify code (SC).

Materials or Tools: The tool, Autostyle, while not immediately available as open source, has multiple components for instructor use (labeling clusters of similar code style) and student use (receiving hints and exemplars towards better code style).

Grading or Assessment: The tool compares submissions to similar clusters that have been pre-labeled by the course responsible. Hints and exemplars linked to that cluster can be used by students to improve their own code quality.

Supported Language(s): Python, Java, Ruby

The greatest strength of this work is to partially automate work to allow human guidance to be used at a large scale with less effort. Although Choudhury et al. [32] only presents an experimental evaluation, the results appear very positive. According to the authors it works for Java, Python and Ruby. Unfortunately, the tool itself does not appear to be accessible to other teachers and the size of exercises was reported to be small (a few lines to tens of lines of code).

As an additional example of independent activity, Bobadilla et al. [20] aims to nudge students towards code quality over time ("SOBO: Nudging Codestyle" card). The bot proposed in their work, *SOBO*, is available on GitHub as open-source software but will require configuration for use in other courses. The main strength and weakness in terms of being used in different courses is that it operates on a GitHub organisation where students keep their code submissions; thus, although it does not require the use of a special tool, it requires for GitHub to be part of the course infrastructure.

In contrast to tools that auto-grade code quality on submission, *SOBO* works on student commits — aimed at providing ongoing formative feedback as work progresses. The limitation on small code examples is removed as *SOBO* works at a repository level and can process all files that seem relevant by file extension. Finally, to avoid the case where instructor-provided code may trigger warnings, *SOBO* uses `git-blame` to only consider code that the students themselves have contributed.

SOBO operates by monitoring student repositories for commits and then analyses the most recent commit for code quality violations (a small subset of Sonar's violations) that are relevant to the course level. However, students may simply push their entire work as a single commit, so other activities, such as helping students plan commits, might be required to encourage students to break their work up into smaller chunks [14].

SOBO: Nudging Codestyle [20]

Description: The tool (SOBO) is a bot that works within a GitHub organisation. As student make code commits, it analyses their submissions using Sonar. Once a violation is found to be recurring an issue is posted with corrective feedback. The message gives a rich description of the violation, along with an example of the violation in another context and its resolution. The tool also has a user interface that allows students to control its activity.

Quality Focus: Using a very small subset of Sonar rules (n=5) that focus on assignment (EL), appropriate implementation (AI) and layout (CL).

Materials or Tools: It is open source, but requires the ability to host as a persistent service and run a class within a GitHub organization.

Grading or Assessment: None

Supported Language(s): Java

Depending upon which violation is most prevalent in the student code, an issue is posted explaining the violation, providing an example and its correction, as well as further reading material. Posting as an issue is both positive in that it uses an authentic pathway for feedback on code rather than an external tool or system, but also negative as it depends upon the student checking their issues.

One final aspect is that *SOBO* provides the students with agency and some entertainment. It has a few basic commands that allow students to interact with the bot, such as disabling and enabling, as well as a few Easter eggs for the more curious students to discover.

7 Discussion

7.1 Research Questions

We start the discussion section by summarising the answers to the research questions formulated in subsection 3.1.

RQ 1: Which code examples related to code quality have been used in the literature on code quality in CS1 education? As expected, the bulk of the examples from the literature use the programming languages currently most widespread in education, namely: *Python*, *Java* and *C/C++*, with a few earlier programs written in *Pascal*. Translating the examples into other Algol-like languages is relatively straightforward. Notable exceptions are some papers adopting block-based environments, usually *Scratch*, or even spreadsheet formulas, to introduce specific forms of code smells and refactoring rules. However, such work was not considered appropriate for a standard CS1 delivery and was therefore out of scope.

As observed in subsection 4.3, most of the retrieved code examples devised explicitly for an introductory context include only a single defect. Approximately 20% of the examples have only one or two lines of code, around 30% three to five lines, another 30% six to ten lines, and the remaining 20% more than ten. It seems likely that such examples were designed to focus on isolated quality issues to reduce the cognitive load imposed by more complex problems.

RQ 1.1: Which topics related to code quality are covered by these examples? A detailed account of the quality topics addressed by the collected examples is summarised in Tables 13 through 23.

A first observation is that roughly half of the "single-defect" examples are about the control flow, predominantly using basic conditional and loop constructs or the connected expressions. Combining these examples with those that introduce issues of data flow, we cover more than two-thirds of the set — corresponding to the *Algorithms* area in Stegeman et al.'s rubric [139]. Higher-level planning aspects involving problem decomposition and program modular structure — the *Structure* area in [139] — are addressed by about 14% of the "single-defect" examples. The remaining 16% concern layout, formatting and documentation issues. The small set of examples exhibiting multiple quality defects in the same program do not introduce additional topics.

Based on the characterisation of types of quality issues introduced in subsection 3.3, the *Missing* qualifier category is covered only by defects affecting the documentation and other minor typographical features (usually only described in words). This should not be surprising since missing elements connected with data, parts of the expressions or the control flow usually means that the program is functionally incorrect.

Examples of missing components in control constructs that could be considered quality defects are, for instance, the lack of a *default* case in a *switch* statement (Ex50), or an empty *catch* block to deal with thrown exceptions (Ex26). In the former case, however, either the listed *switch* cases cover all the possible values of the control expression, in which case the *default* is redundant, or the execution could give rise to runtime errors. Similarly, in the latter case, when the exceptions are not properly caught, the code execution will break off abruptly. Even if using such components is suggested as a temporary solution during debugging, this makes only sense for potentially incorrect code.

We can further observe that most issues involving expressions are due to redundancies (*Unnecessary* category), with only two more examples that present instances of avoidable duplication and scarce clarity. *Duplication* and *Unnecessary* together amount to more than 70% of the examples in the *Control/Block* class, whereas the *Inconsistent* category is not represented. This may be a consequence of the small size of most examples.

At a finer-grained level, the types of defects addressed in more (at least three) different sources are:

- *Data*: Global variables used with local scope and useless or unnecessary variables;
- *Expression*: Unnecessary evaluations of conditions;
- *Control/Block*: Subsequent conditionals with the same (single) branch, duplicated code that can be "factored" between conditional branches (this case being very frequent), unnecessary *if* statements that can be collapsed into a Boolean expression (e.g., to return), subsequent *if* statements that can be condensed in a single construct using an *else* clause, nested conditionals that can be condensed using Boolean operators, ineffective statements, inappropriate loop choice;
- *Organisation*: Issues due to lack of decomposition;
- *Documentation*: Meaningless or misleading choice of names and direct use of literals instead of symbolic constants;
- *Typographic*: Messy layout and formatting.

Section 7.2 elaborates on the characterisation of the covered topics from the perspective of Kirk et al.'s [88] more abstract taxonomy.

RQ 2: Which teaching activities related to code quality have been used in the literature on code quality in CS1 education? As discussed in subsection 4.4, we identified 28 instructional activities appropriate for the CS1 level. Four of them are meant to be mainly conducted by the instructor, seven imply some significant interaction between teacher and students, and all the others are proposed as essentially independent tasks for the learner. This indicates that most of the activities reported in the literature rely upon the student both making use of a tool and being able to interpret any feedback they receive.

Additionally, we did not observe significant connections between the activities and the catalogued examples, except in a few cases where the paper makes explicit reference to a specific defect and its improvement (e.g., [20, 32, 67]). This may be because comprehensive lists of examples intended for an introductory level were unavailable until very recently. Our work aims to make further progress in this respect.

RQ 2.1: Which topics related to code quality are covered by these activities? To characterise the topics addressed by the activities identified in this study, summarised in Table 11, we refer to Kirk et al.'s quality principles [88]. Nine of them cover a broad range of code quality issues, but are loosely related to Kirk et al.'s taxonomy, in that they make use of static analysis tools that implement a variety of checks (e.g., [17]) or draw from textbooks and other resources available in the specific context (e.g., [31]).

The other activities target specific quality principles, sometimes multiple different principles. The counts are shown in Table 24. The four most commonly covered categories were: *Explanatory Language (EL)*, *Clear Layout (CL)*, *Simple Constructs (SC)*, and *Modular*

Table 24: Counts of quality principles covered by the identified CS1 activities.

Code	Taxonomy Item	Count
EL	Explanatory language	8
CL	Clear layout	7
SC	Simple Constructs	8
CD	Consistent Design	1
NC	Non-redundant Content	1
AI	Appropriate Implementation	3
AD	Avoid Duplication	3
MS	Modular Structure	10
	Unspecified	9

Structure (MS). For instance, Hundhausen et al. [61] report on a facilitated peer review activity that was scaffolded by providing students with a checklist of specific questions spanning three of Kirk et al.'s categories (*EL*, *CL* and *MS*). In Keen and Mammen [79], students completed a term-long project in which program decomposition into modular structure (*MS*) is emphasized in the milestones. In Nascimento et al. [111], the authors piloted a custom-built tool, *IQCheck*, that makes use of natural language processing to evaluate the quality of identifier names (*EL*) of student code, taking into account the text of the problem specification.

7.2 Teaching Quality Principles

Kirk et al. [88] focus on a set of principles that are intended to inform the teaching of code style — specifically helping teachers and students to better articulate the implications of code quality decisions (i.e., why some approaches are recommended over others). We will briefly review each principle and link it to the curated set of defects listed in section 5. Moreover, in Table 9 we summarise the relationships between Kirk et al.'s categories and typical quality indicators identified in the literature. For a detailed discussion of the rationale underlying each principle, the reader is referred to Kirk et al. [88].

7.2.1 Explanatory Language (EL). This principle specifies that *'The intent and meaning of code is explicit'*. This includes the appropriate use of comments to explain code as needed, but also the use of descriptive identifiers for symbolic constants, variables, functions and methods. Code that is well constructed and uses meaningful variable and function names is sometimes described as "self-documenting" because the intent and meaning of the code are evident, reducing the need for comments. Examples from Table 22 include:

- Identifiers should be meaningful and describe the purpose of the data or function they refer to *(DC2)*.
- Header comments that are necessary to explain the purpose of the code should be included where appropriate *(DC4)*.
- 'Magic' numbers should be replaced with symbolic constants to make the meaning of the numbers explicit *(DC3)*.

7.2.2 Clear Layout (CL). This principle specifies that *'Different elements are easy to distinguish, and the relationships between them are apparent'*. The focus of this principle is the layout and formatting of the content. Although modern IDE support has improved formatting and layout, the principle remains relevant. The use of layout implies a deliberate choice by the programmer to imply relationships within code. In the absence of an expressive layout, the reader can potentially be misled about the relationships between code elements, making the code more difficult to understand. Examples from Table 22 include:

- Long lines of code, or comments, that are difficult to read *(TP1, TP3)*.
- Failure to include blank lines between blocks that would separate the blocks of code more obviously, or too many blank lines that disrupt the reading flow *(TP4, TP5)*.
- Using spaces between some operators and not others. Poor, or inconsistent use of spaces can unintentionally imply that elements are related when they are not *(TP5)*.

7.2.3 Simple Constructs (SC). This principle specifies that *'Coding constructs are implemented in a way that minimises complexity for the intended reader'*. The principle of Simple Constructs most obviously applies to control flow but can also be applied to expressions that should be simplified where possible. Adhering to this principle is especially important when students are still learning to read and understand code. Examples from the tables in section 5 include:

- A complex expression should be simplified using Boolean and relational operators *(EX4)*.
- *if* statements should have the simplest structure that aligns with the desired control flow *(CB2)*.
- Nested *if* statements that can be simplified using a Boolean operator instead *(CB8)*.
- Using a nested *if* inside an else branch when an *elif/else-if* is possible *(CB21)*.

7.2.4 Consistent Design (CD). This principle captures the benefits of code conventions by specifying *'Elements that are similar in nature are presented and used in a similar way'*. Consistency is important because code requires less mental effort to understand when it adheres to standard patterns. This consistency is important within the code, but also within a broader community, which may have adopted *conventions* that standardise layout or other components of code use according to a set of (largely arbitrary) rules. Examples of standardisation or consistent design from the tables in section 5 include:

- Code that does not adhere to naming conventions is harder to read because it defies expectations of the reader *(TP6)*.
- Missing components, such as header comments, that are required by code conventions *(DC4)*.
- The types of variables should be consistent when used for the same purpose *(DT10)*.
- The ordering of parameters should be consistent to avoid simple mistakes occurring when functions or methods are called *(OR7)*.

7.2.5 Non-redundant Content (NC). This principle specifies that *'All elements that are introduced are meaningfully used'*. It should be noted that redundancy here focuses exclusively on code (and comments) that do not contribute meaningfully to the program; redundancy related to duplication is dealt with separately. Examples of this category from the tables in section 5 include:

- Comments that do not meaningfully convey additional information are redundant and should be removed (*DC1*).
- Code that is never executed is considered redundant and should be removed (*CB7*).
- Code that can be removed without impacting the functional correctness of the code is also redundant (*EX1, EX2, DT1, DT2, DT3, CB3, CB4, CB6*, and *OR1*).

7.2.6 Appropriate Implementation (AI). This principle specifies: *'Implementation choices are suited to the problem to be solved'*. In the context of CS1, this can be interpreted as "using the right programming constructs for the task at hand". This includes algorithmic issues where the code is more complex than needed because of poor implementation choices (e.g., adding a sentinel to a list of data values during a read process, then needing to remove it later). Examples from the tables in section 5 include:

- The appropriate type should be used for each variable, e.g., integers should not be used in place of a Boolean *(DT7)*.
- Functions that modify parameters passed by reference should instead return values where it makes sense to do so *(DT12)*.
- The control variable in a *for* loop should not be modified within the loop *(CB15)*.
- A *while* loop should not be used as a single-iteration replacement for an *if* statement *(CB18)*.
- Loop constructs should be selected based on the purpose of the loop — *for* loops should be used to iterate through a data structure, and *while* loops should be used when the end condition is indeterminate using static analysis *(CB20)*.
- Infinite loops with a break to terminate the loop should be replaced with a loop using an explicit end condition *(CB22)*.

7.2.7 Avoid Duplication (AD). This principle specifies that: *'Code duplication is avoided'*.[4] Duplicated code requires more effort to read and makes it more likely to be inconsistent and to introduce errors. Examples from Tables 16 and 18 include:

- Expressions that could be calculated once and stored for reuse in multiple places *(EX3)*.
- Similar code appearing in multiple branches of conditional statements *(CB10, CB12)*.
- Conditional statements with the same condition that could be merged *(CB11)*.

7.2.8 Modular Structure (MS). This principle states that *'Related code is grouped together and dependencies between groups minimised'*. This is intended to apply to algorithmic structures, functions, and classes (where appropriate). If code that is related is grouped together then it is easier to understand the relationships between elements, and if the inter-relatedness of 'chunks' of code is minimized, then the mental effort required to remember the relationships is reduced. Further, modular code is easier to modify without requiring consequential changes to other code. This is a common design principle that is typically introduced in CS1 courses, and aligns with advice to minimise the use of global variables and instead prefer variables with local scope. Examples from Tables 14, 21 and 23 include:

- Parameters should be passed by value whenever possible, rather than passed by reference, to improve the independence of functions *(DT9)*.
- Long lists of parameters should be made more compact to reduce high levels of inter-relatedness between code (i.e., to improve modularity) *(DT13)*.
- Where possible, variables should be local rather than global (or instance variables if teaching OOP) *(DT8, DT14)*
- Decomposition should be used to reduce the complexity of problems *(*OR6).
- Long functions should be decomposed into smaller parts *(OR8)*.

7.3 Modelling the Cognitive Demands of Code Quality-Related Tasks

One of the concerns when teaching to novice programmers is to identify the cognitive effort implied by the tasks we propose. This is of course related to the level of complexity and abstraction of the code artefacts learners are required to interact with.

A reference framework that has been proved useful to analyse core aspects of program comprehension is Schulte's *Block Model* [129]. This instrument introduces a two-dimensional mapping of abstraction and complexity (see Figures 3–6), primarily addressed to introductory programming education. The merits of the Block Model have been substantiated by different studies. In particular, it can be used to characterise programming tasks [72, 127]. According to Whalley and Kasto [157], it has also the potential to support a more accurate categorisation than those provided by Bloom's [7] or the SOLO [15] learning taxonomies.

The horizontal dimension of the Block Model distinguishes among different levels of abstraction to look at the artefact at hand: from *static* syntactic features of the program (*Text surface*), to *dynamic* aspects pertaining to the working of the underlying *notional machine* [44] (*Program execution*), to the *intent*-related purposes of the program as a problem-solving tool (*Function/purpose*). The vertical dimension spans through four levels of structural complexity of program components: atomic components such as expressions and individual statements (*Atoms*), code chunks built up by combining basic flow control constructs (*Blocks*), relations and dependencies between program parts (*Relationships*), and overall program structure (*Macrostructure*).

The two dimensions outlined above are organised into a 4×3 matrix, as shown in the figures 3–6, where the rows represent a hierarchy of increasingly complex programming structures, whereas the columns correspond to the three levels of abstraction through which we inspect the code. From a pedagogical point of view, stepping upwards within the matrix, from simple to complex, and rightwards, from text to execution to purpose, is connected with progress in the program comprehension process [129, 130] and implies thinking at increasingly higher levels of abstraction.

Drawing inspiration from Izu et al. [72], we can then take a similar approach and map code quality-related tasks into different cells of the matrix. In this respect, Schulte's framework can also be used to identify paths in the Block Model corresponding to issue-fixing or refactoring interventions — more specifically, trajectories from the cell pertaining to detection of a quality issue to the cell accounting for how the artefact can be restructured. A similar

[4] In this paper, we use the term *Avoid Duplication* instead of *Avoid Repetition*, as in [88], for the same principle to ensure that the reader does not interpret this as avoiding the use of loops.

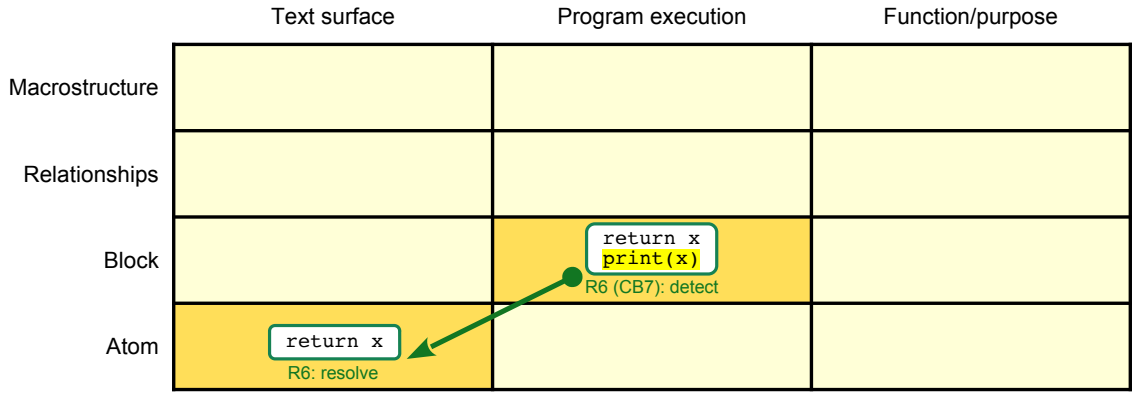


Figure 3: Mappings of sample code quality tasks into Schulte's Block Model framework. The example is an instance of unreachable code (R6).

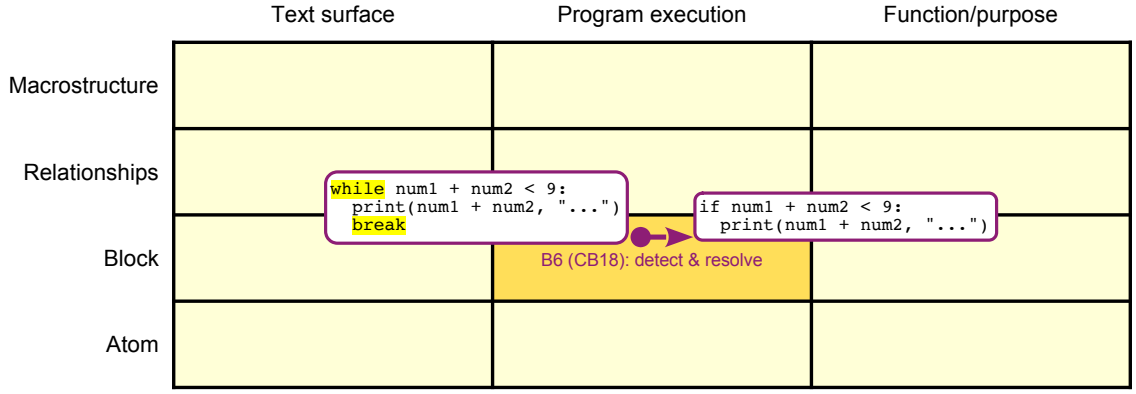


Figure 4: Mappings of sample code quality tasks into the Block Model matrix: an instances of "single-iteration loop" (B6).

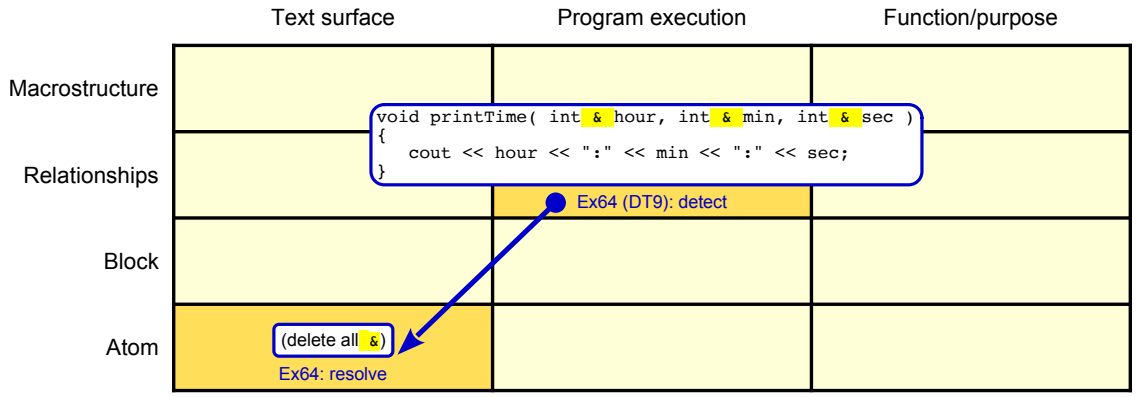


Figure 5: Mappings of sample code quality tasks into the Block Model matrix: an instance of inappropriate parameter passing (Ex64).

picture gives some insight into the level of challenge students may face. A few examples follow (refer to the categorisation in section 5):

- Several quality issues pertaining to the syntactic facet can be detected and fixed at the same structural level of the *Text surface* area — e.g., layout and formatting issues such as *TP1*, *TP2* or *TP5*. It is also the area where automatic tools are more helpful. In the spirit of the Block Model, the prominent focus

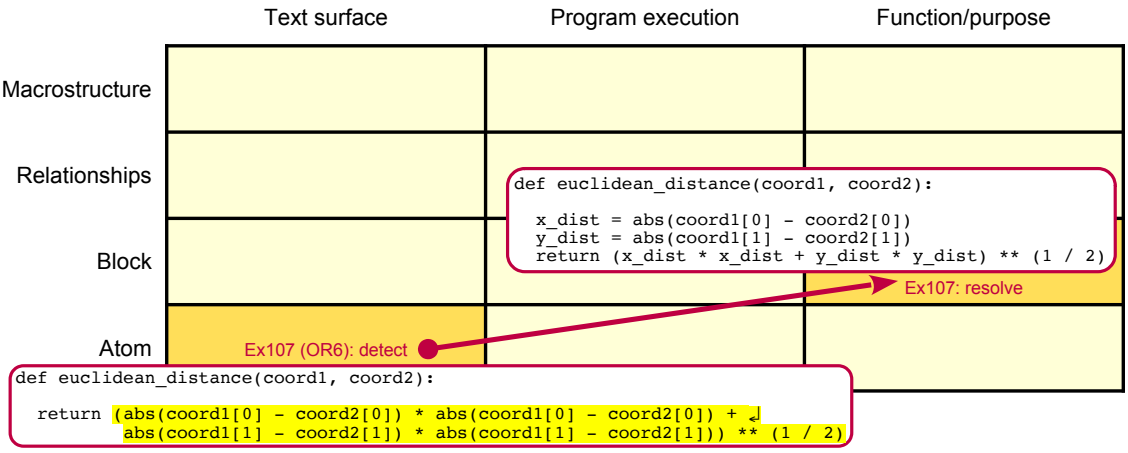


Figure 6: Mappings of sample code quality tasks into the Block Model matrix: example of lack of decomposition (Ex107).

of this area is on code layout and indentation, since they are related to the learner's awareness of code structure.

- Duplicate code is detected at a *Text surface* level, typically either *Atom* or *Block*, but can be resolved at a *Program execution* level — examples are *CB10*, *CB11*, *CB12* at the *Block* level; *CB13* at the *Relationships* level; or *OR3* (extract function) at the *Macrostructure* level.
- Lack of conciseness may emerge at a *Text surface* level, but the code can be cleaned at a corresponding level of the *Program execution* dimension — e.g., *EX1*, *EX4* at the *Atom* level or *CB2*, *CB8* at the *Block* level. In the case of long scripts, such as *OR8*, the target code can be even the result of a revision at the *Macrostructure* level.
- Useless or unreachable code, on the other hand, is often detected at a *Program execution* (*Atom*, *Block* or *Relationships*) level, but once identified it can be removed at the *Text surface* (usually *Block*) level; related example are *DT1*, *DT3*, *CB4* and *CB7*.
- Well-structuredness of code — including examples such as *CB15*, *CB16*, *CB20*, *CB21* and *CB22* — is commonly evaluated and enhanced at the *Block* level of *Program execution*.
- Decomposition can involve knowledge extrinsic to the program code, in which case it can be evaluated and dealt with at the *Function/purpose* (up to *Macrostructure*) level; examples are *OR2*, *OR5* and *OR6*.
- Similarly, header and inline comments usually imply extrinsic considerations which make only sense in the context of the problem to be solved and hence pertain to the *Function/purpose* facet of the model.

For the sake of illustration, the figures 3 through 6 consider four examples drawn from our catalogue. R6 is a simple instance of unreachable code (Figure 3): to identify this issue it is necessary to envisage how a chunk of code is executed (*Program execution – Block* cell) and to fix it it is sufficient to remove a single statement (*Text surface – Atom*). The second example, B6 (Figure 4), is about the inappropriate use of a loop to perform a single iteration, whereas a simpler *if* statement would do the job more smoothly. This kind of issue can be detected and resolved "within" the same (*Program execution – Block*) cell.

The third example, Ex64 (Figure 5), presents a case where using pass-by-reference parameters is not a sensible choice: this kind of issue can be recognised by putting into relation the list of parameters in the function header with the code in the function body and, more in general, also with the caller code (*Program execution – Relationships*); fixing the issue is then effortless (*Text surface – Atom*). The last example, Ex107 (Figure 6), includes a long expression (*Text surface – Atom*) of no immediate interpretation: to render it more readable it is necessary to decompose the computation — into a code chunk — in such a way that the meaning of its parts becomes clearer, i.e., by making self-evident the purpose of each sub-expression (*Function/purpose – Block*).

As visualised in Figures 3 and 5, it is interesting to observe that a refactoring intervention, i.e., editing the code, is not always more cognitively demanding than the detection of that quality issue. There are defects, such as R6 and Ex64, that once identified are indeed straightforward to resolve. Others, such as instances of code duplication or Ex107, are easily detected but improving the code may be challenging for novices.

Dealing with code presenting multiple quality defects can be described by more complex "trajectories" along the Block Model dimensions. The four refactoring steps illustrated in Figure 1, for instance, can be mapped into two edits of expressions at the *Atom* level and two interventions on chunks of code at the *Block* level, all of them moving from *Text surface* (detection) to *Program execution*, where the transformation can be understood — see Figure 7.

As a further observation, in a few cases, such as those pertaining to CB8 or CB10, it may appear that the refactoring consists simply in the application of syntactic rules, and is therefore restricted to the left column of the Block Model matrix. Nevertheless, to *understand* the rationale underlying the transformations and be able to appreciate the equivalence between the code before and after editing, the student is required to reason at the *Program execution* level.

To sum up, Schulte's Block Model can be an helpful device available to the instructor in order to plan a sequence of code quality

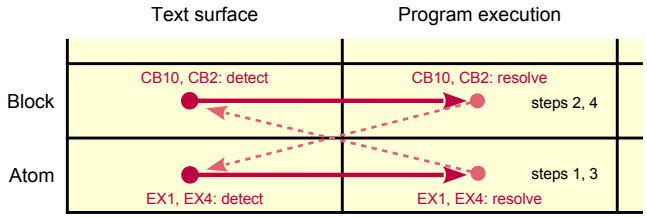


Figure 7: Mapping into the Block Model framework of the task described in Figure 1, dealing with with multiple code quality defects in the same program (E1).

tasks of gradually increasing complexity, starting from examples that can be mapped into the bottom-left cells of the matrix and then moving forward, towards higher structural complexity and more abstract perspectives on the programs. The (very) simplest tasks we can think of are restricted to detection of issues identifiable by just looking at the program text, or resolvable by removing a few lines of code after receiving feedback on the kind of issue (possibly via automatic tools). A reasonable objective for CS1, however, could be to cover (at least) a bit of each of the four complexity levels of the *Program execution* perspective.

7.4 Challenges and Recommendations for CS1 Instructors

Fitting code quality in CS curricula is challenging because it requires a shift in emphasis on the code and the coding process. This challenge is reflected in many ways: (1) the lack of topics in the curricula that would support code quality activities, such as reading, reviewing and modifying code; (2) the lack of focused instruction to develop related programming skills; (3) the lack of validated assessments of these programming skills, for example, about how to pick a programming construct for a specific problem; and, maybe most importantly, (4) the perception that teaching code quality would be an additional burden to our already crammed CS curricula. We now discuss each challenge and provide some recommendations for educators.

Challenge 1: Limited quality-related topics in the CS curricula. Historically, CS curricula emphasise concepts over coding. Coding is viewed as a vehicle for demonstrating CS concepts. Therefore, unsurprisingly the way our students engage with code during their formal education years does not match how they do it in the profession. A finding from Börstler et al.'s 2017 ITiCSE Working Group [24] revealed a stark contrast between the programming tasks of students and developers. Though the sample size of the survey was small, 58% of students *disagreed* with the statement that other people are reading or modifying the code that they have written, whereas only 27% of developers disagreed with that statement [24].

This challenge is also reflected in the limited richness of activities that we found in the literature. Many of these activities are inherently designed as a supplement — often a resource or tool that is offered to students outside of class with no direct instruction component in lecture (refer to section 4, point 4.4.2 for more details). In addition, most activities are focused on the end product, such as identifying code style violations in students' code and providing feedback or illustrative examples (as summarised in Table 12), and rarely on the refactoring process. Lastly, most papers lack suggestions for assessment and grading of similar activities (see section 4, point 4.4.3), what presents further issues of incentivising students as we discuss later in this section.

> Recommendation 1: Replace some writing code activities with alternative activities that require reading, reviewing and modifying code.

In our view, CS1 instructors should reflect on how code quality fits in their current curriculum, and how to make small and gradual changes to their classroom activities. For example, instead of giving students time to write a few lines of code in a workshop, you may provide one or two alternative programs to compare and evaluate, which they can actively discuss using a *Think, Pair, Share* structure. Replacing code creation activities with code evaluation activities would not only change the focus to code quality, but it would be a very valuable task considering the increasing use of AI-supported coding tools.

Challenge 2: Limited incentives for students to write quality code. Our analysis revealed that CS1 students have limited incentives to write quality code. Two separate observations illustrate this issue. First, very few activities actually reward good code quality in terms of grading. Our extraction of pedagogical activities in subsection 4.4 showed that only 18% of those addressed to the CS1 level include any grading components related to code quality. Instead, the majority of the activities were tools that students could use to improve the quality of their code, but had little incentive to do so.

Second, there is a lack of activities in the CS1 curricula that would enable good habits to improve code style, in particular code review activities. Given our framing of code quality in terms of ease of understanding and modifying code, the limited opportunities for students to write code that will be read or modified by others certainly reduce the intrinsic motivations for caring about code quality. Indeed, the activities we extracted in the present study show that only three (11%) of the 28 activities involved *peer review*, which would require students to read each other's code.

> Recommendation 2: Engage students by adding incentives for evaluating the quality of their own code, as well as of the code developed by their peers.

Similar incentives could range from opportunities to interact with code written by other programmers or peers, group discussion of code quality for participation marks, peer reviews or, when possible, adding a code quality grading component to practical assignments. The instructor could direct peer reviews to relevant areas by creating a one-page checklist with a small set of defects to identify and fix; for example, if the assignment focuses on iteration, the checklist could include some of the defects related to the use of variables and loop constructs, plus some typographic defects. Later assignments can then have an overlapping set of defects to check. The same approach could be used to provide a self-checklist to be completed before the last submission.

Whatever the "operational" incentives we may figure out, it is clear that students' *intrinsic* motivation to learn about code quality

is the most effective one. While discussing the challenges faced by novices to learn programming, du Boulay [44] identified as the first area of difficulty the "general problem of *orientation*, finding out what programming is for [...] and what the eventual advantages might be of expending effort in learning the skill" [44, p. 57]. In other words: how to make programming *relevant* to learners from *their* perspective. Similarly, when trying to teach students to be aware of and appreciate the value and benefit of code quality, we have to make this program-related topic *meaningful* to them, from a learner's perspective.

Challenge 3: Limited time to cover code quality topics. Covering all quality principles and the whole range of code quality defects is a time-intensive endeavour. Most CS1 courses are dense in content, and there is already a challenge to make time for new content. Instructors should then become familiar with the Kirk et al.'s taxonomy and explore the catalogue to identify a few quality principles and a manageable subset of defects.

Students write complicated and verbose code. This is often due to inappropriate choice of constructs combined with duplication and redundancy. Duplication is a perfect entry point for discussion of code structure as it is easy to spot and fix. Redundancy is sometimes harder to spot, but it forces students to comprehend the program flow. The refactoring involved to fix duplication and eliminate redundancy helps students to improve their implementation choices; furthermore, avoiding duplication motivates the need for modularity.

> Recommendation 3: Introduce code quality to CS1 students by focusing first and foremost on two quality principles: *Non-redundant Content* and *Avoid Duplication*.

These two principles target code structure, providing students with a sense of achievement as they learn to detect and remove unnecessary and verbose code. They could be complemented with other principles — we suggest focusing attention on *Explanatory Language* in the first half of the course and *Modular Structure* in the second half.

Challenge 4: Limited incentives for instructors to adopt quality-oriented activities. Switching to the instructor's perspective, there are also limited incentives to adopt activities in terms of the effort required to find and adapt those drawn from the literature. Whilst we have found multiple papers that focus on an activity, we also found a new challenge in extracting the actual activity in a reusable way for an instructor to consider.

Due to the expectations and constraints of research papers, more room is given to the experimental process and the outcomes than to the actual steps taken by an instructor and the experience of the students. Consequently, repetition in a new context is difficult if not impossible. This further complicates matters as the activities are hardly ever evaluated in a different context. As it appears from the activities we have identified, the range of contexts varies considerably and effort is needed to repeat activities more broadly to test their potential. (refer to section 4, point 4.4.2 for more details).

We have suggested that developing activity cards might be a more viable way of communicating the essence of an activity — see section 6. Whilst the initial sample of such cards in [70] (Appendix B) is not extensive, our hope is that an open collection, much like *Nifty Assignments* [115] or the Canterbury Question Bank [128], could be a starting point in increasing the incentives for instructors to easily find and more readily adopt code quality-related activities for their students.

> Recommendation 4: Is addressed to the broader CS Education community in terms of an invitation to share their efforts to develop code quality-related activities in ways that facilitate portability and set-up in different contexts.

8 Summary and Conclusion

While the bulk of studies addressing introductory programming are concerned with typical misconceptions and errors made by novices, a distinctive perspective of the present work is that we have restricted our attention to syntactically and functionally correct code, which may exhibit quality defects, such as code smells or poor documentation.

The CS Education literature has recently identified gaps regarding the coverage of code quality in the CS curriculum and advocates for the development of related teaching materials. At the same time, in the last decade, we have witnessed a growing research effort to explore, capture and analyse both the quality of student code and their perceptions about this topic, as accounted for by Keuning et al.'s recent systematic review [85]. In an attempt to provide suggestions for introducing code quality at the early stages of programming education, this working group analysed 248 papers to produce two resources intended for use at the CS1 level: a catalogue of code examples that illustrate quality defects and a representative set of instructional activities.

After Keuning et al.'s review, two very recent works by Silva et al. [133] and Řechtáčková et al. [123] have developed a catalogue of code smells with a broader scope. The decisions of what issues to catalogue may have been influenced by multiple factors, such as course structure, programming language and assumed program size. Besides, reporting every little issue may overload students and discourage them from progressing.

A key contribution of the present study is the curation of a catalogue of examples that builds upon prior efforts and captures most structural code defects that have a simple-to-describe resolving transformation, both from the instructor's perspective as well as from the student's perspective. The set of examples is large and has been organised into the 63 patterns presented in subsection 4.3 (with 161 examples to illustrate several variations). The patterns have been arranged according to two dimensions: source and issue qualifier. This is meant to facilitate the planning of activities focusing on a single dimension, for example, by choosing a source such as *Data* or *Organization*, or perhaps an issue such as *Unnecessary* or *Duplication*. The catalogue is extensive, but it cannot be all-comprehensive. We hope to consolidate this initial effort into a public repository that can evolve and grow with consensus from CS1 educators.

In addition, we have identified from the literature 28 instructional activities that address code quality at the CS1 level, including an analysis of their scope. This has also led to the proposal of "activity

cards" to summarise the main characteristics of code quality-related activities from the perspective of an interested instructor. More work needs to be done to facilitate the dissemination and adoption of successful activities.

To sum up, the material covered in this work is intended to support CS1 instructors and is meant to offer:

- A broad repertory of examples among which to choose, based on the learning objectives and the course context (full set can be found in [70] under Appendix A;
- A multifaceted characterisation of the examples supporting the identification of the topics to address (section 5);
- A set of insightful activities to draw inspiration from if planning some significant intervention (section 6 and longer set in [70] under Appendix B);
- A framework to characterise code quality topics from an abstract perspective (subsection 4.2 and subsection 7.2);
- The suggestion of a model to relate code quality to program comprehension and characterise the challenge level of the considered tasks (subsection 7.3).

We hope that such composite resources could support the adoption and further development of teaching materials by CS1 instructors willing to address code quality in their classrooms.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1984. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA. First edition.

[2] Felix Adler, Gordon Fraser, Eva Gründinger, Nina Körber, Simon Labrenz, Jonas Lerchenberger, Stephan Lukasczyk, and Sebastian Schweikl. 2021. Improving Readability of Scratch Programs with Search-based Refactoring. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, NY, 120–130. https://doi.org/10.1109/SCAM52516.2021.00023

[3] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Järvinen. 2004. Supporting Students in C++ Programming Courses with Automatic Program Style Assessment. *JITE* 3 (01 2004), 245–262. https://doi.org/10.28945/300

[4] Ella Albrecht and Jens Grabowski. 2020. Sometimes It's Just Sloppiness - Studying Students' Programming Errors and Misconceptions. 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, 340–345. https://doi.org/10.1145/3328778.3366862

[5] Francisco Alfredo, André L. Santos, and Nuno Garrido. 2022. Sprinter: A Didactic Linter for Structured Programming. In *Third International Computer Programming Education Conference (ICPEC 2022) (Open Access Series in Informatics (OASIcs), Vol. 102)*, Alberto Simões and João Carlos Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:8. https://doi.org/10.4230/OASIcs.ICPEC.2022.2

[6] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2024. Automating Source Code Refactoring in the Classroom. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 60–66. https://doi.org/10.1145/3626252.3630787

[7] Lorin W. Anderson and David R. Krathwohl (Eds.). 2001. *Taxonomy for Learning, Teaching and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman, New York.

[8] Raul Andrade and João Brunet. 2018. Can students help themselves? An investigation of students' feedback on the quality of the source code. In *2018 IEEE Frontiers in Education Conference (FIE)*. IEEE, NY, 1–8. https://doi.org/10.1109/FIE.2018.8658503

[9] Eliane Araujo, Dalton Serey, and Jorge Figueiredo. 2016. Qualitative aspects of students' programs: Can we make them measurable?. In *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, NY, 1–8. https://doi.org/10.1109/FIE.2016.7757725

[10] Pasquale Ardimento, Mario Luca Bernardi, and Marta Cimitile. 2020. Software Analytics to Support Students in Object-Oriented Programming Tasks: An Empirical Study. *IEEE Access* 8 (2020), 132171–132187. https://doi.org/10.1109/ACCESS.2020.3010172

[11] Yu Bai, Tao Wang, and Huaimin Wang. 2019. Amelioration of Teaching Strategies by Exploring Code Quality and Submission Behavior. *IEEE Access* 7 (2019), 152744–152754. https://doi.org/10.1109/ACCESS.2019.2948640

[12] Gergő Balogh. 2015. Comparison of Software Quality in the Work of Children and Professional Developers Based on Their Classroom Exercises. In *Computational Science and Its Applications – ICCSA 2015*, Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Marina L. Gavrilova, Ana Maria Alves Coutinho Rocha, Carmelo Torre, David Taniar, and Bernady O. Apduhan (Eds.). Springer International Publishing, Cham, 36–46.

[13] Elisa Baniassad, Ivan Beschastnikh, Reid Holmes, Gregor Kiczales, and Meghan Allen. 2019. Learning to listen for design. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 179–186. https://doi.org/10.1145/3359591.3359738

[14] Amanda Berg, Simon Osnes, and Richard Glassey. 2022. If in Doubt, Try Three: Developing Better Version Control Commit Behaviour with First Year Students. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 362–368. https://doi.org/10.1145/3478431.3499371

[15] J. B. Biggs and K. F Collis. 1982. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, New York, USA.

[16] Anastasiia Birillo, Elizaveta Artser, Yaroslav Golubev, Maria Tigina, Hieke Keuning, Nikolay Vyahhi, and Timofey Bryksin. 2023. Detecting Code Quality Issues in Pre-Written Templates of Programming Tasks in Online Courses. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 152–158. https://doi.org/10.1145/3587102.3588800

[17] Anastasiia Birillo, Ilya Vlasov, Artyom Burylov, Vitalii Selishchev, Artyom Goncharov, Elena Tikhomirova, Nikolay Vyahhi, and Timofey Bryksin. 2022. Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments. In *Proc. of the 53rd ACM Tech. Symposium on Computer Science Education* (Providence, RI, USA) *(SIGCSE 2022)*. ACM, New York, NY, USA, 307–313.

[18] Jeremiah Blanchard, John R. Hott, Vincent Berry, Rebecca Carroll, Bob Edmison, Richard Glassey, Oscar Karnalim, Brian Plancher, and Seán Russell. 2022. Stop Reinventing the Wheel! Promoting Community Software in Computing Education. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (Dublin, Ireland) *(ITiCSE-WGR '22)*. Association for Computing Machinery, New York, NY, USA, 261–292. https://doi.org/10.1145/3571785.3574129

[19] Hannah Blau and J. Eliot B. Moss. 2015. FrenchPress Gives Students Automated Feedback on Java Program Flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (Vilnius, Lithuania) *(ITiCSE '15)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/2729094.2742622

[20] Sofia Bobadilla, Richard Glassey, Alexandre Bergel, and Martin Monperrus. 2024. SOBO: A Feedback Bot to Nudge Code Quality in Programming Courses. *IEEE Software* 41, 2 (2024), 68–76. https://doi.org/10.1109/MS.2023.3298729

[21] Jürgen Börstler, Kwabena E Bennin, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, Bonnie MacKellar, Rodrigo Duran, Harald Störrle, Daniel Toll, et al. 2023. Developers talking about code quality. *Empirical Software Engineering* 28, 6 (2023), 128.

[22] Jürgen Börstler, Michael E. Caspersen, and Marie Nordström. 2016. Beauty and the Beast: on the readability of object-oriented example programs. *Software Quality Journal* 24, 2 (2016), 231–246. https://doi.org/10.1007/s11219-015-9267-5

[23] Jürgen Börstler, Marie Nordström, and James H. Paterson. 2011. On the Quality of Examples in Introductory Java Textbooks. *Trans. Comput. Educ.* 11, 1, Article 3 (Feb. 2011), 21 pages. http://doi.acm.org/1921607.1921610

[24] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2018. "I Know It When I See It" Perceptions of Code Quality: ITiCSE '17 Working Group Report. In *Proc. of the 2017 ITiCSE Conference on Working Group Reports* (Bologna, Italy) *(ITiCSE-WGR '17)*. ACM, New York, NY, USA, 70–85.

[25] Dennis M. Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring Static Quality of Student Code. 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, 13–17. https://doi.org/10.1145/1999747.1999754

[26] Richard K. Brewer. 1976. Documentation standards for beginning students. In *Proceedings of the ACM SIGCSE-SIGCUE Technical Symposium on Computer Science and Education (SIGCSE '76)*. Association for Computing Machinery, New York, NY, USA, 69–73. https://doi.org/10.1145/800107.803450

[27] Aline Brito, Andre Hora, and Marco Tulio Valente. 2022. Understanding Refactoring Tasks over Time: A Study Using Refactoring Graphs. In *Anais do XXV Congresso Ibero-Americano em Engenharia de Software* (Córdoba). SBC, Porto

Alegre, RS, Brasil, 330–344. https://doi.org/10.5753/cibse.2022.20982
[28] Brandon Carlson. 2008. An Agile Classroom Experience: Teaching TDD and Refactoring. In *Agile 2008 Conference*. IEEE, NY, 465–469. https://doi.org/10.1109/Agile.2008.39
[29] Charis Charitsis, Chris Piech, and John C. Mitchell. 2022. Function Names: Quantifying the Relationship Between Identifiers and Their Functionality to Improve Them. In *Proceedings of the Ninth ACM Conference on Learning @ Scale* (New York City, NY, USA) *(L@S '22)*. Association for Computing Machinery, New York, NY, USA, 93–101. https://doi.org/10.1145/3491140.3528269
[30] Hsi-Min Chen, Wei-Han Chen, and Chi-Chen Lee. 2018. An Automated Assessment System for Analysis of Coding Convention Violations in Java Programming Assignments. *Journal of Information Science and Engineering* 34, 5 (Sep 2018), 1203–1221. https://doi.org/10.6688/JISE.201809_34(5).0006
[31] Alexandru Chirvase, Laura Ruse, Mihnea Muraru, Mariana Mocanu, and Vlad Ciobanu. 2021. Clean Code - Delivering A Lightweight Course. In *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*. IEEE, NY, 420–423. https://doi.org/10.1109/CSCS52396.2021.00075
[32] Rohan Roy Choudhury, HeZheng Yin, Joseph Moghadam, and Armando Fox. 2016. AutoStyle: Toward Coding Style Feedback At Scale. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion* (San Francisco, California, USA) *(CSCW '16 Companion)*. Association for Computing Machinery, New York, NY, USA, 21–24. https://doi.org/10.1145/2818052.2874315
[33] Wellesley College. 2000. *Simplifying Boolean Expressions and Conditionals*. http://cs111.wellesley.edu/~cs111/archive/cs111_spring00/public_html/lectures/boolean-simplification.html
[34] Yania Crespo, Arturo Gonzalez-Escribano, and Mario Piattini. 2021. Carrot and Stick approaches revisited when managing Technical Debt in an educational context. In *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, NY, 99–108. https://doi.org/10.1109/TechDebt52882.2021.00020
[35] Carlos Dantas, Adriano Rocha, and Marcelo Maia. 2023. Assessing the Readability of ChatGPT Code Snippet Recommendations: A Comparative Study. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering* (Campo Grande, Brazil) *(SBES '23)*. Association for Computing Machinery, New York, NY, USA, 283–292. https://doi.org/10.1145/3613372.3613413
[36] Pedro Henrique de Andrade Gomes, Rogério Eduardo Garcia, Gabriel Spadon, Danilo Medeiros Eler, Celso Olivete, and Ronaldo Celso Messias Correia. 2017. Teaching software quality via source code inspection tool. In *2017 IEEE Frontiers in Education Conference (FIE)*. IEEE, NY, 1–8. https://doi.org/10.1109/FIE.2017.8190658
[37] Eustáquio São José De Faria, Juan Manuel Adán-Coello, and Keiji Yamanaka. 2006. Forming Groups for Collaborative Learning in Introductory Computer Programming Courses Based on Students' Programming Styles: An Empirical Study. In *Proceedings. Frontiers in Education. 36th Annual Conference*. IEEE, NY, 6–11. https://doi.org/10.1109/FIE.2006.322313
[38] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, and Nasser Giacaman. 2018. Unencapsulated Collection: A Teachable Design Smell. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 332–337. https://doi.org/10.1145/3159450.3159469
[39] 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
[40] Tomche Delev and Dejan Gjorgjevikj. 2017. Static analysis of source code written by novice programmers. In *2017 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, NY, 825–830. https://doi.org/10.1109/EDUCON.2017.7942942
[41] Linus Dietz, Johannes Manner, Simon Harrer, and Jörg Lenhard. 2018. Teaching Clean Code. In *SE-WS 2018: Software Engineering Workshops 2018* (Ulm, Germany) *(CEUR Workshop Proceedings, Vol. 2066)*. ceur-ws.org, Ulm, Germany, 24–27. https://ceur-ws.org/Vol-2066/ Combined Proceedings of the Workshops of the German Software Engineering Conference 2018 (SE 2018); Ulm, Germany, March 6, 2018.
[42] Niels Doorn, Tanja Vos, Beatriz Marín, and Erik Barendsen. 2023. Set the right example when teaching programming: Test Informed Learning with Examples (TILE). In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, NY, 269–280. https://doi.org/10.1109/ICST57152.2023.00033
[43] Hoyama Maria dos Santos, Vinicius H. S. Durelli, Maurício Souza, Eduardo Figueiredo, Lucas Timoteo da Silva, and Rafael S. Durelli. 2019. CleanGame: Gamifying the Identification of Code Smells. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering* (Salvador, Brazil) *(SBES '19)*. Association for Computing Machinery, New York, NY, USA, 437–446. https://doi.org/10.1145/3350768.3352490
[44] Benedict du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.
[45] 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)*. ACM, New York, NY, USA, 65–73. https://doi.org/10.1145/3105726.3106182
[46] Davide Falessi and Philippe Kruchten. 2015. Five Reasons for Including Technical Debt in the Software Engineering Curriculum. In *Proceedings of the 2015 European Conference on Software Architecture Workshops* (Dubrovnik, Cavtat, Croatia) *(ECSAW '15)*. Association for Computing Machinery, New York, NY, USA, Article 28, 4 pages. https://doi.org/10.1145/2797433.2797462
[47] Amin Milani Fard and Ali Mesbah. 2013. JSNOSE: Detecting JavaScript Code Smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, NY, 116–125. https://doi.org/10.1109/SCAM.2013.6648192
[48] 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.
[49] Ann E. Fleury. 2001. Encapsualtion and reuse as viewed by java students. In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education* (Charlotte, North Carolina, USA) *(SIGCSE '01)*. Association for Computing Machinery, New York, NY, USA, 189–193. https://doi.org/10.1145/364447.364582
[50] Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, USA.
[51] Nobuo Funabiki, Takuya Ogawa, Nobuya Ishihara, Minoru Kuribayashi, and Wen-Chung Kao. 2016. A Proposal of Coding Rule Learning Function in Java Programming Learning Assistant System. In *2016 10th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*. IEEE, NY, 561–566. https://doi.org/10.1109/CISIS.2016.94
[52] Iris Gaber and Amir Kirsh. 2018. The Effect of Reporting Known Issues on Students' Work. In *Proc. of the 49th ACM Tech. Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. ACM, New York, NY, USA, 74–79.
[53] Iris Gaber and Amir Kirsh. 2021. Using Examples as Guideposts for Programming Exercises: Providing Support while Preserving the Challenge. In *16th International Conference on Computer Science & Education (ICCSE)*. IEEE, NY, 391–397. https://doi.org/10.1109/ICCSE51940.2021.9569541
[54] Nupur Garg and Aaron W. Keen. 2018. Earthworm: Automated Decomposition Suggestions. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '18)*. Association for Computing Machinery, New York, NY, USA, Article 16, 5 pages. https://doi.org/10.1145/3279720.3279736
[55] Pedro Henrique Gomes, Rogério Eduardo Garcia, Danilo Medeiros Eler, Ronaldo Celso Correia, and Celso Olivete Junior. 2021. Software Quality as a Subsidy for Teaching Programming. In *2021 IEEE Frontiers in Education Conference (FIE)*. IEEE, NY, 1–9. https://doi.org/10.1109/FIE49875.2021.9637475
[56] Wouter Groeneveld, Dries Martin, Tibo Poncelet, and Kris Aerts. 2022. Are Undergraduate Creative Coders Clean Coders? A Correlation Study. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 314–320. https://doi.org/10.1145/3478431.3499345
[57] Rowan Hart, Brian Hays, Connor McMillin, El Kindi Rezig, Gustavo Rodriguez-Rivera, and Jeffrey A. Turkstra. 2023. Eastwood-Tidy: C Linting for Automated Code Style Assessment in Programming Courses. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 799–805. https://doi.org/10.1145/3545945.3569817
[58] Hiroaki Hashiura, Saeko Matsuura, and Seiichi Komiya. 2010. A tool for diagnosing the quality of java program and a method for its effective utilization in education. In *Proceedings of the 9th WSEAS International Conference on Applications of Computer Engineering* (Penang, Malaysia) *(ACE'10)*. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 276–282.
[59] Felienne Hermans and Efthimia Aivaloglou. 2016. Do code smells hamper novice programming? A controlled experiment on Scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, NY, 1–10. https://doi.org/10.1109/ICPC.2016.7503706
[60] Zhewei Hu and Edward F. Gehringer. 2019. Improving Feedback on GitHub Pull Requests: A Bots Approach. In *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, NY, 1–9. https://doi.org/10.1109/FIE43999.2019.9028685
[61] Christopher Hundhausen, Anukrati Agrawal, Dana Fairbrother, and Michael Trevisan. 2009. Integrating Pedagogical Code Reviews into a CS1 Course: An Empirical Study. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA) *(SIGCSE '09)*. Association for Computing Machinery, New York, NY, USA, 291–295. https://doi.org/10.1145/1508865.1508972
[62] Callum Iddon, Nasser Giacaman, and Valerio Terragni. 2023. GRADESTYLE: GitHub-Integrated and Automated Assessment of Java Code Style. In *2023*

IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET). IEEE, NY, 192–197. https://doi.org/10.1109/ICSE-SEET58685.2023.00024

[63] Essa Imhmed, Edgar Ceh-Varela, Hashim Abu-Gellban, and Scott Kilgore. 2024. Fostering Code Quality Practices Among Undergraduate Novice Programmers. *J. Comput. Sci. Coll.* 39, 7 (may 2024), 21–32.

[64] Yuki Ito, Atsuo Hazeyama, Yasuhiko Morimoto, Hiroaki Kaminaga, Shoichi Nakamura, and Youzou Miyadera. 2014. A Method for Detecting Bad Smells and ITS Application to Software Engineering Education. In *2014 IIAI 3rd International Conference on Advanced Applied Informatics*. IEEE, NY, 670–675. https://doi.org/10.1109/IIAI-AAI.2014.139

[65] Cruz Izu. 2022. Modelling the Use of Abstraction in Algorithmic Problem Solving. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 193–199. https://doi.org/10.1145/3502718.3524758

[66] Cruz Izu and Shrey Chandra. 2022. Are We There Yet? Novices' Code Smells linked to Loop Constructs. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1151. https://doi.org/10.1145/3478432.3499064

[67] Cruz Izu, Paul Denny, and Sayoni Roy. 2022. A Resource to Support Novices Refactoring Conditional Statements. In *Proc. of the 27th ACM Conf. on Innovation and Technology in Computer Science Education* (Dublin, Ireland) *(ITiCSE '22)*. ACM, New York, NY, USA, 344–350.

[68] Cruz Izu and Claudio Mirolo. 2023. Exploring CS1 Student's Notions of Code Quality. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 12–18. https://doi.org/10.1145/3587102.3588808

[69] Cruz Izu and Claudio Mirolo. 2024. Asking Students to Refactor their Code: A Simple and Valuable Exercise. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 73–79. https://doi.org/10.1145/3649217.3653546

[70] Cruz Izu and Claudio Mirolo. 2024. Introduction to Code Quality at CS1 Level: Examples and Activities (v1). https://doi.org/10.5281/zenodo.14258104

[71] Cruz Izu and Claudio Mirolo. 2024. Towards Comprehensive Assessment of Code Quality at CS1-level: Tools, Rubrics and Refactoring Rules. In *2024 IEEE Global Engineering Education Conference (EDUCON)* (Kos Island, Greece). IEEE, NY, 1–10. https://doi.org/10.1109/EDUCON60312.2024.10578672

[72] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and et al. 2019. Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories. 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, 27–52. https://doi.org/10.1145/3344429.3372501

[73] Julian Jansen, Ana Oprescu, and Magiel Bruntink. 2017. The Impact of Automated Code Quality Feedback in Programming Education. In *SATToSE 2017: Seminar on Advanced Techniques and Tools for Software Evolution* (Madrid, Spain) *(CEUR Workshop Proceedings, Vol. 2070)*. ceur-ws.org, Madrid, Spain, 19 pages. https://ceur-ws.org/Vol-2070/ Post-proceedings of the Tenth Seminar on Advanced Techniques and Tools for Software Evolution.

[74] Lucy Jiang, Robert Rewcastle, Paul Denny, and Ewan Tempero. 2020. CompareCFG: Providing Visual Feedback on Code Quality Using Control Flow Graphs. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 493–499. https://doi.org/10.1145/3341525.3387362

[75] Saj-Nicole A. Joni and Elliot Soloway. 1986. But My Program Runs! Discourse Rules for Novice Programmers. *Journal of Educational Computing Research* 2, 1 (1986), 95–125. https://doi.org/10.2190/6E5W-AR7C-NX76-HUT2

[76] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying Student Misconceptions of Programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, Wisconsin, USA) *(SIGCSE '10)*. ACM, New York, NY, USA, 107–111. https://doi.org/10.1145/1734263.1734299

[77] Oscar Karnalim and Simon. 2021. Promoting Code Quality via Automated Feedback on Student Submissions. In *2021 IEEE Frontiers in Education Conference (FIE)*. IEEE, NY, 1–5. https://doi.org/10.1109/FIE49875.2021.9637193

[78] Remin Kasahara, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. 2019. Applying Gamification to Motivate Students to Write High-Quality Code in Programming Assignments. In *Proc. of the 2019 ACM Conf. on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. ACM, New York, NY, USA, 92–98.

[79] Aaron Keen and Kurt Mammen. 2015. Program Decomposition and Complexity in CS1. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 48–53. https://doi.org/10.1145/2676723.2677219

[80] Brian W Kernighan and P. J. Plauger. 1978. *The elements of programming style, 2nd edition*. McGraw-Hill, NY.

[81] Hieke Keuning. 2020. *Automated Feedback for Learning Code Refactoring*. Ph. D. Dissertation. Open Universiteit. https://research.ou.nl/en/publications/automated-feedback-for-learning-code-refactoring

[82] 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

[83] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student Refactoring Behaviour in a Programming Tutor. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3428029.3428043

[84] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A Tutoring System to Learn Code Refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 562–568. https://doi.org/10.1145/3408877.3432526

[85] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2023. A Systematic Mapping Study of Code Quality in Education. In *Proc. of the 2023 Conf. on Innovation and Technology in Computer Science Education* (Turku, Finland) *(ITiCSE 2023)*. ACM, New York, NY, USA, 5–11.

[86] Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. 2020. On Assuring Learning About Code Quality. In *Proceedings of the Twenty-Second Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE'20)*. Association for Computing Machinery, New York, NY, USA, 86–94. https://doi.org/10.1145/3373165.3373175

[87] Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. 2022. Teaching Code Quality in High School Programming Courses - Understanding Teachers' Needs. In *Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. ACM, New York, NY, USA, 36–45.

[88] Diana Kirk, Andrew Luxton-Reilly, and Ewan Tempero. 2024. A Literature-Informed Model for Code Style Principles to Support Teachers of Text-Based Programming. In *Proceedings of the 26th Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE '24)*. Association for Computing Machinery, New York, NY, USA, 134–143. https://doi.org/10.1145/3636243.3636258

[89] Diana Kirk, Ewan Tempero, Andrew Luxton-Reilly, and Tyne Crow. 2020. High School Teachers' Understanding of Code Style. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/3428029.3428047

[90] Amruth N. Kumar. 2022. Refactoring Examples. https://problets.org/courses/opl/refactoring/index.html.

[91] Mario Leyva. 2023. *Refactoring Tutor: An IDE Integrated Tool for Practicing Key Techniques to Refactor Code*. Master's thesis. Master of Engineering, Massachusetts Institute of Technology.

[92] David Liu, Jonathan Calver, and Michelle Craig. 2024. A Static Analysis Tool in CS1: Student Usage and Perceptions of PythonTA. In *Proceedings of the 26th Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE '24)*. Association for Computing Machinery, New York, NY, USA, 172–181. https://doi.org/10.1145/3636243.3636262

[93] 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

[94] 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 2020)*. Association for Computing Machinery, New York, NY, USA, 56–60. https://doi.org/10.1145/3385209.3385226

[95] Carlos López, Jesús M. Alonso, Raúl Marticorena, and Jesús M. Maudes. 2014. Design of e-activities for the learning of code refactoring tasks. In *2014 International Symposium on Computers in Education (SIIE)*. IEEE, NY, 35–40. https://doi.org/10.1109/SIIE.2014.7017701

[96] Shaoxiao Lu, Xu Wang, Haici Zhou, Qing Sun, Wenge Rong, and Ji Wu. 2021. Anomaly Detection for Early Warning in Object-oriented Programming Course. In *2021 IEEE International Conference on Engineering, Technology & Education (TALE)*. IEEE, NY, 01–08. https://doi.org/10.1109/TALE52509.2021.9678677

[97] Nikola Luburić, Dragan Vidaković, Jelena Slivka, Simona Prokić, Katarina-Glorija Grujić, Aleksandar Kovačević, and Goran Sladić. 2022. Clean Code Tutoring: Makings of a Foundation. In *Proceedings of the 14th International Conference on Computer Supported Education - Volume 1: CSEDU*. INSTICC, SciTePress,

Setúbal, Portugal, 137–148. https://doi.org/10.5220/0010800900003182

[98] Roope Luukkainen, Jussi Kasurinen, Uolevi Nikula, and Valentina Lenarduzzi. 2022. ASPA: A static analyser to support learning and continuous feedback on programming courses. An empirical validation. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Software Engineering Education and Training* (Pittsburgh, Pennsylvania) *(ICSE-SEET '22)*. Association for Computing Machinery, New York, NY, USA, 29–39. https://doi.org/10.1145/3510456.3514149

[99] Yuzhi Ma and Eli Tilevich. 2021. "You have said too much": Java-like verbosity anti-patterns in Python codebases. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E* (Chicago, IL, USA) *(SPLASH-E 2021)*. Association for Computing Machinery, New York, NY, USA, 13––18. https://doi.org/10.1145/3484272.3484960

[100] S. Mäkelä and V. Leppänen. 2004. Japroch: A tool for checking programming style. In *Proceedings of the Koli Calling International Conference on Computing Education Research* (Kolin Kolistelut-Koli Calling, Finland) *(Koli Calling 2004)*. Espoo : Helsinki University of Technology, Koli, Finland, 151.

[101] M.V. Mäntylä. 2005. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, NY, 10 pages. https://doi.org/10.1109/ISESE.2005.1541837

[102] M. Mäntylä, J. Vanhanen, and C. Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, NY, 381–384. https://doi.org/10.1109/ICSM.2003.1235447

[103] Kirby McMaster, Samuel Sambasivam, and Stuart Wolthuis. 2013. Teaching programming style with ugly code. In *Proceedings of the Information Systems Educators Conference*, Vol. 2167. Education Special Interest Group of the AITP, San Antonio, Texas, USA, 1435.

[104] Kirby McMaster, Samuel Sambasivam, and Stuart Wolthuis. 2014. Software development using C++: beauty-and-the-beast. *Issues in Informing Science and Information Technology* 11 (2014), 073–084.

[105] Susan A. Mengel and Vinay Yerramilli. 1999. A Case Study of the Static Analysis of the Quality of Novice Student Programs. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education* (New Orleans, Louisiana, USA) *(SIGCSE '99)*. Association for Computing Machinery, New York, NY, USA, 78–82. https://doi.org/10.1145/299649.299689

[106] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2024. Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Trans. Comput. Educ.* 24, 1, Article 10 (feb 2024), 43 pages. https://doi.org/10.1145/3636515

[107] G. Michaelson. 1996. Automatic analysis of functional program style. In *Proceedings of 1996 Australian Software Engineering Conference.* IEEE, NY, 38–46. https://doi.org/10.1109/ASWEC.1996.534121

[108] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering* 36, 1 (2010), 20–36. https://doi.org/10.1109/TSE.2009.50

[109] Mustafa Alrifaee Mohammad Abdallah. 1970. A Heuristic Tool for Measuring Software Quality Using Program Language Standards. *The International Arab Journal of Information Technology (IAJIT)* 19, 03 (1970), 314–322. https://doi.org/10.34028/iajit/19/3/4

[110] Arthur-Jozsef Molnar, Simona Motogna, and Cristina Vlad. 2020. Using static analysis tools to assist student project evaluation. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Education through Advanced Software Engineering and Artificial Intelligence* (Virtual, USA) *(EASEAI 2020)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3412453.3423195

[111] Marcos Nascimento, Eliane Araújo, Dalton Serey, and Jorge Figueiredo. 2020. The Role of Source Code Vocabulary in Programming Teaching and Learning. In *2020 IEEE Frontiers in Education Conference (FIE)* (Uppsala). IEEE Press, NY, 1–8. https://doi.org/10.1109/FIE44824.2020.9274137

[112] Eduardo Oliveira, Hieke Keuning, and Johan Jeuring. 2023. Student Code Refactoring Misconceptions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 19–25. https://doi.org/10.1145/3587102.3588840

[113] Paul W. Oman and Curtis R. Cook. 1991. A programming style taxonomy. *Journal of Systems and Software* 15, 3 (1991), 287–301. https://doi.org/10.1016/0164-1212(91)90044-7

[114] J. Walker Orr. 2022. Automatic assessment of the design quality of student python and java programs. *J. Comput. Sci. Coll.* 38, 1 (nov 2022), 27–36.

[115] Nick Parlante, Julie Zelenski, Eric S. Roberts, Jed Rembold, Ben Stephenson, Jonathan Hudson, Stephanie Valentine, Juliette Woodrow, Kathleen Creel, Nick Bowman, Larry "Joshua" Crotts, Andrew Matzureff, and Mike Izbicki. 2022. Nifty Assignments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1067–1068. https://doi.org/10.1145/3478432.3499268

[116] Harrie Passier, Sylvia Stuurman, and Harold Pootjes. 2014. Beautiful JavaScript: how to guide students to create good and elegant code. In *Proceedings of the Computer Science Education Research Conference* (Berlin, Germany) *(CSERC '14)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2691352.2691358

[117] James Perretta, Westley Weimer, and Andrew DeOrio. 2019. Human vs. Automated Coding Style Grading in Computing Education. In *Proc. of the 126th Annual ASEE Conference*. American Society for Engineering Education, DC, 12 pages. https://doi.org/10.18260/1-2--32906

[118] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An Empirical Study of Iterative Improvement in Programming Assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 410–415. https://doi.org/10.1145/2676723.2677279

[119] Bernard John Poole and Timothy S. Meyer. 1996. Implementing a set of guidelines for CS majors in the production of program code. *SIGCSE Bull.* 28, 2 (jun 1996), 43–48. https://doi.org/10.1145/228296.228304

[120] Yuliia Prokop, Olena Trofymenko, and Olexander Zadereyko. 2023. Developing students' code style skills. In *2023 IEEE 18th International Conference on Computer Science and Information Technologies (CSIT)*. IEEE, NY, 1–4. https://doi.org/10.1109/CSIT61576.2023.10324182

[121] Lin Qiu and Christopher Riesbeck. 2008. An incremental model for developing educational critiquing systems: Experiences with the Java Critiquer. *Jl. of Interactive Learning Research* 19, 1 (2008), 119–145.

[122] Sarnath Ramnath and Brahma Dathan. 2008. Evolving an integrated curriculum for object-oriented analysis and design. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 337–341. https://doi.org/10.1145/1352135.1352252

[123] Anna Řechtáčková, Radek Pelánek, and Tomáš Effenberger. 2024. Catalog of Code Quality Defects in Introductory Programming. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 59–65. https://doi.org/10.1145/3649217.3653638

[124] P.A. Relf. 2005. Tool assisted identifier naming for improved software readability: an empirical study. In *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, NY, 10 pp.–. https://doi.org/10.1109/ISESE.2005.1541814

[125] Sally S. Robinson and M. L. Soffa. 1980. An instructional aid for student programs. In *Proceedings of the Eleventh SIGCSE Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '80)*. Association for Computing Machinery, New York, NY, USA, 118–129. https://doi.org/10.1145/800140.804623

[126] Dean Sanders and Janet Hartman. 1987. Assessing the quality of programs: a topic for the CS2 course. In *Proceedings of the Eighteenth SIGCSE Technical Symposium on Computer Science Education* (St. Louis, Missouri, USA) *(SIGCSE '87)*. Association for Computing Machinery, New York, NY, USA, 92–96. https://doi.org/10.1145/31820.31741

[127] Kate Sanders, Marzieh Ahmadzadeh, Tony Clear, Stephen H. Edwards, Mikey Goldweber, Chris Johnson, Raymond Lister, Robert McCartney, Elizabeth Patitsas, and Jaime Spacco. 2013. The Canterbury QuestionBank: Building a Repository of Multiple-choice CS1 and CS2 Questions. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports* (Canterbury, England, United Kingdom) *(ITiCSE -WGR '13)*. ACM, New York, NY, USA, 33–52. https://doi.org/10.1145/2543882.2543885

[128] Kate Sanders, Marzieh Ahmadzadeh, Tony Clear, Stephen H. Edwards, Mikey Goldweber, Chris Johnson, Raymond Lister, Robert McCartney, Elizabeth Patitsas, and Jaime Spacco. 2013. The Canterbury QuestionBank: building a repository of multiple-choice CS1 and CS2 questions. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-Working Group Reports* (Canterbury, England, United Kingdom) *(ITiCSE -WGR '13)*. Association for Computing Machinery, New York, NY, USA, 33–52. https://doi.org/10.1145/2543882.2543885

[129] Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535

[130] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. 2010. An Introduction to Program Comprehension for Computer Science Educators. In *Proceedings of the 2010 ITiCSE Working Group Reports* (Ankara, Turkey) *(ITiCSE-WGR '10)*. ACM, New York, NY, USA, 65–86. https://doi.org/10.1145/1971681.1971687

[131] Rika Sekimoto and Kenji Kaijiri. 2000. A Diagnosis System of Programming Styles Using Program Patterns. *IEICE Transactions on Information and Systems* E83-D, 4 (04 2000), 722–728.

[132] Allyson Senger, Stephen H. Edwards, and Margaret Ellis. 2022. Helping Student Programmers Through Industrial-Strength Static Analysis: A Replication Study. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3478431.3499310

[133] Eryck Pedro da Silva, Ricardo Caceffo, and Rodolfo Azevedo. 2023. When Test Cases Are Not Enough: Identification, Assessment, and Rationale of Misconceptions in Correct Code (MC^3). *Brazilian Journal of Computers in Education* 31 (Dec. 2023), 1165–1199. https://doi.org/10.5753/rbie.2023.3552

[134] Suzanne Smith, Sara Stoecklin, and Catharina Serino. 2006. An innovative approach to teaching refactoring. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, Texas, USA) *(SIGCSE '06)*. Association for Computing Machinery, New York, NY, USA, 349–353. https://doi.org/10.1145/1121341.1121451

[135] Suzanne Smith, Sara F. Stoecklin, and Judy Mullins. 2004. Taking cohesion into the classroom. *J. Comput. Sci. Coll.* 20, 2 (dec 2004), 296–303.

[136] E. Soloway. 1986. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (sep 1986), 850–858. https://doi.org/10.1145/6592.6594

[137] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive Strategies and Looping Constructs: An Empirical Study. *Commun. ACM* 26, 11 (Nov. 1983), 853–860. https://doi.org/10.1145/182.358436

[138] Sai Krishna Sripada and Y. Raghu Reddy. 2015. Code Comprehension Activities in Undergraduate Software Engineering Course - A Case Study. In *2015 24th Australasian Software Engineering Conference*. IEEE, NY, 68–77. https://doi.org/10.1109/ASWEC.2015.18

[139] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2014. Towards an Empirically Validated Model for Assessment of Code Quality. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '14)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/2674683.2674702

[140] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a Rubric for Feedback on Code Quality in Programming Courses. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '16)*. Association for Computing Machinery, New York, NY, USA, 160–164. https://doi.org/10.1145/2999541.2999555

[141] Sara Stoecklin, Suzanne Smith, and Catharina Serino. 2007. Teaching students to build well formed object-oriented methods through refactoring. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (Covington, Kentucky, USA) *(SIGCSE '07)*. Association for Computing Machinery, New York, NY, USA, 145–149. https://doi.org/10.1145/1227310.1227364

[142] Ivan Tan and Christopher M. Poskitt. 2024. Fixing Your Own Smells: Adding a Mistake-Based Familiarisation Step When Teaching Code Refactoring. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 1307–1313. https://doi.org/10.1145/3626252.3630856

[143] Stuti Tandon, Vijay Kumar, and V.B. Singh. 2024. Study of Code Smells: A Review and Research Agenda. *International Journal of Mathematical, Engineering and Management Sciences* 9, 3 (2024), 472–498. https://doi.org/10.33889/IJMEMS.2024.9.3.025

[144] Peeratham Techapalokul and Eli Tilevich. 2017. Understanding recurring quality problems and their impact on code sharing in block-based software. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, NY, 43–51. https://doi.org/10.1109/VLHCC.2017.8103449

[145] Veronika Thurner. 2019. Fostering the Comprehension of the Object-Oriented Programming Paradigm by a Virtual Lab Exercise. In *2019 5th Experiment International Conference (exp.at'19)*. IEEE, NY, 137–142. https://doi.org/10.1109/EXPAT.2019.8876484

[146] Sirazum Munira Tisha, Rufino A. Oregon, Gerald Baumgartner, Fernando Alegre, and Juana Moreno. 2023. An automatic grading system for a high school-level computational thinking course. In *Proceedings of the 4th International Workshop on Software Engineering Education for the Next Generation* (Pittsburgh, Pennsylvania) *(SEENG '22)*. Association for Computing Machinery, New York, NY, USA, 20–27. https://doi.org/10.1145/3528231.3528357

[147] 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* (<conf-loc>, <city>Dunedin</city>, <country>New Zealand</country>, </conf-loc>) *(ACE '04)*. Australian Computer Society, Inc., AUS, 317–325.

[148] Leo C Ureel II. 2020. *Critiquing Antipatterns In Novice Code*. Ph. D. Dissertation. Michigan Technological University.

[149] Leo C. Ureel II and Charles Wallace. 2019. Automated Critique of Early Programming Antipatterns. 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, 738–744. https://doi.org/10.1145/3287324.3287463 ISECON 2012.

[150] Ashok Kumar Veerasamy, Daryl D'Souza, and Mikko-Jussi Laakso. 2016. Identifying Novice Student Programming Misconceptions and Errors From Summative Assessments. *Journal of Educational Technology Systems* 45, 1 (2016), 50–73. https://doi.org/10.1177/0047239515627263

[151] Leslie Waguespack. 2008. Hammers, Nails, Windows, Doors and Teaching Great Design. *Information Systems Education Journal* 6, 45 (2008), 3–18. http://isedj.org/6/45/

[152] Leslie Waguespack. 2013. A Design Quality Learning Unit in Relational Data Modeling Based on Thriving Systems Properties. *Information Systems Education Journal* 11, 4 (2013), 18–30. http://isedj.org/2013-11/ A preliminary version appears in The Proceedings of ISECON 2012.

[153] Leslie Waguespack. 2015. A Design Quality Learning Unit in OOa Modeling Bridging the Engineer and the Artist. *Information Systems Education Journal* 13, 1 (2015), 58–70. http://isedj.org/2015-13/

[154] Tomoyoshi Wakabayashi, Shinpei Ogata, and Saeko Matsuura. 2011. Dependency analysis for learning class structure for novice Java programmer. In *2011 IEEE 2nd International Conference on Software Engineering and Service Science*. IEEE, NY, 532–535. https://doi.org/10.1109/ICSESS.2011.5982370

[155] Yanqing Wang. 2011. Research and Practice on Education of SQA at Source Code Level. *International Journal of Engineering Education* 27 (01 2011), 70–76.

[156] Jacqueline Whalley, Tony Clear, Phil Robbins, and Errol Thompson. 2011. Salient elements in novice solutions to code writing problems. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114* (Perth, Australia) *(ACE '11)*. Australian Computer Society, Inc., AUS, 37–46.

[157] Jacqueline Whalley and Nadia Kasto. 2013. Revisiting Models of Human Conceptualisation in the Context of a Programming Examination. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136* (Adelaide, Australia) *(ACE '13)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 67–76. http://dl.acm.org/citation.cfm?id=2667199.2667207

[158] Eliane Wiese, Anna N. Rafferty, and Jordan Pyper. 2022. Readable vs. Writable Code: A Survey of Intermediate Students' Structure Choices. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 321–327. https://doi.org/10.1145/3478431.3499413

[159] Eliane S. Wiese, Anna N. Rafferty, and Armando Fox. 2019. Linking Code Readability, Structure, and Comprehension among Novices: It's Complicated. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training* (Montreal, Quebec, Canada) *(ICSE-SEET '19)*. IEEE Press, NY, 84–94. https://doi.org/10.1109/ICSE-SEET.2019.00017

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

[161] Juliette Woodrow, Ali Malik, and Chris Piech. 2024. AI Teaches the Art of Elegant Coding: Timely, Fair, and Helpful Style Feedback in a Global Course. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 1442–1448. https://doi.org/10.1145/3626252.3630773

[162] Wei Xu, Chao Wu, and Jianliang Lu. 2021. Exploration of Experimental Teaching Reforms on C Programming Design Course. In *2021 International Symposium on Advances in Informatics, Electronics and Education (ISAIEE)*. IEEE, NY, 330–333. https://doi.org/10.1109/ISAIEE55071.2021.00086

[163] Susilo Veri Yulianto and Inggriani Liem. 2014. Automatic grader for programming assignment using source code analyzer. In *2014 International Conference on Data and Software Engineering (ICODSE)*. IEEE, NY, 1–4. https://doi.org/10.1109/ICODSE.2014.7062687

[164] Marsha Zaidman. 2004. Teaching defensive programming in Java. *J. Comput. Sci. Coll.* 19, 3 (2004), 33–43.

[165] Imre Zsigmond., Maria Iuliana Bocicor., and Arthur-Jozsef Molnar. 2020. Gamification based Learning Environment for Computer Science Students. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. INSTICC, SciTePress, Setúbal, Portugal, 556–563. https://doi.org/10.5220/0009579305560563