



CSM: A Code Style Model for Computing Educators

DIANA KIRK, ANDREW LUXTON-REILLY, and EWAN TEMPERO, School of Computer Science, University of Auckland, Auckland, New Zealand

Objectives Code style is an important aspect of text-based programming because programs written with good style are considered easier to understand and change and so improve the maintainability of the delivered software product. However teaching code style is complicated by the existence of many style guides and standards that contain inconsistent, low-level advice. The objective of the study is to support educators in helping students understand the nature of these inconsistencies by highlighting the rationale behind rules and their contextual nature.

Participants Seventy members of local and national computing science education (CSE) groups contributed in the exploratory phase. Fifty-two members of the ACM SIGCSE members mailing list and the Australasian Chapter contributed in the evaluation and refinement phases. These mailing lists represent a large, global population of CSE educators.

Study Methods We created, evaluated and refined an abstract code style model (CSM) comprising a *Definition* and eight *Principles* for code style. For CSM creation, we took an exploratory approach based on an examination of the existing literature on code quality and style. To evaluate the level of community endorsement for the model, we analysed quantitative and qualitative data from an international survey questionnaire. We refined the CSM based on an analysis of the qualitative survey data.

Findings Analysis of the data from the survey indicated a strong level of support for the CSM. Quantitative data revealed a general agreement with the *Principles*, with responses for five *Principles* achieving a minimum of 84 percent agreement and all *Principles* achieving a minimum of 66 percent agreement. Qualitative data contained very few comments that voiced dissatisfaction with a basic aspect of the model. We found that barriers to developing a community-wide CSM were the strongly-held beliefs about style held by some and the abstract nature of the CSM.

Conclusions The creation of an abstract set of *Principles* for code style is generally perceived by the community to be worthwhile and important and our approach has high levels of endorsement. A validated version of the CSM has been created.

CCS Concepts: • **Social and professional topics** → **Computing education**;

Additional Key Words and Phrases: computing education, text-based programming, software quality, code style, online survey

Authors' Contact Information: Diana Kirk (corresponding author), School of Computer Science, University of Auckland, Auckland, New Zealand; e-mail: diana.kirk@auckland.ac.nz; Andrew Luxton-Reilly, School of Computer Science, University of Auckland, Auckland, New Zealand; e-mail: a.luxton-reilly@auckland.ac.nz; Ewan Tempero, School of Computer Science, University of Auckland, Auckland, New Zealand; e-mail: e.tempero@auckland.ac.nz.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1946-6226/2025/4-ART6

<https://doi.org/10.1145/3716861>

ACM Reference format:

Diana Kirk, Andrew Luxton-Reilly, and Ewan Tempero. 2025. CSM: A Code Style Model for Computing Educators. *ACM Trans. Comput. Educ.* 25, 1, Article 6 (April 2025), 39 pages. <https://doi.org/10.1145/3716861>

1 Introduction

In addition to learning a chosen programming language, students are advised to learn what it means to write code with ‘good style’. This is important both for students collaborating on projects and for professional developers tasked with writing code that can easily be maintained by others. Educators would like to prepare students for working in the industry, where developers are expected to write code with ‘good style’. The importance of style within the industry is evidenced in the many available style guides and standards, each of which offers advice and guidance to programmers within a specific community. However, these vary in content and offer conflicting advice and this leaves many educators struggling to explain to students what style is, why it matters and why one style rule is better than another. We believe that in order to teach code style effectively, some means are needed to reconcile the apparent inconsistencies and lack of clarity in existing style guides. We have created a **Code Style Model (CSM)** as a means to do so.

In this article, we present the *CSM*. We describe how we created, evaluated and refined the model. Our aim is that the *CSM* will support educators in helping students understand these inconsistencies and will provide a foundation on which to base code-style discussions.

The difficulty faced by educators and students regarding code style is that, while there is a great deal of advice given, there is often no rationale for why the advice should be followed. This is particularly a problem when the advice appears inconsistent. For example, the PEP-8 Style Guide for Python Code [54] says ‘Avoid extraneous whitespace in [...] immediately inside parentheses, brackets or braces’ and then later ‘Always surround [...] binary operators with a single space on either side...’. While the two cases are clearly delineated, how should a teacher respond to the question ‘Why are spaces ok in one situation and not the other?’ Complicating the picture is that immediately following the advice about whitespace around operators is, ‘If operators with different priorities are used, consider adding whitespace around the operators’! There is clearly some underlying rationale for this different advice. We argue that to best serve educators and students, the focus should be on the rationale, rather than specific rules. The *CSM* is intended to capture the rationale for the different advice.

Experienced educators will be familiar with the issue and have developed their own satisfactory responses, but it would have taken significant effort. They may also find themselves having to change their response in different contexts, such as for different languages (e.g., Google advises 4-space tabs for Python [30] but 2-space tabs for Java [29]). Furthermore, the responses are likely to be subtly different between different educators, so students may be exposed to many different ideas about what style is without understanding the underlying rationale behind the disparate style rules. Furthermore, inexperienced educators must develop their own responses with little in the way of support. The *CSM* provides a framework for organising the different advice.

The *CSM* comprises a *Definition* and a set of overarching *Principles* that capture the underlying aspects of code style at a conceptual level. For ‘code style’, we consider quality attributes that are important for product maintainability. The attributes cited for maintainability in the product quality models relate to understanding, changing and testing code. For style, we restrict our model to understanding and changing, i.e., we do not consider the testability of code. We have defined style as ‘*aspects of software quality that can be determined by looking at the source code, constrained*

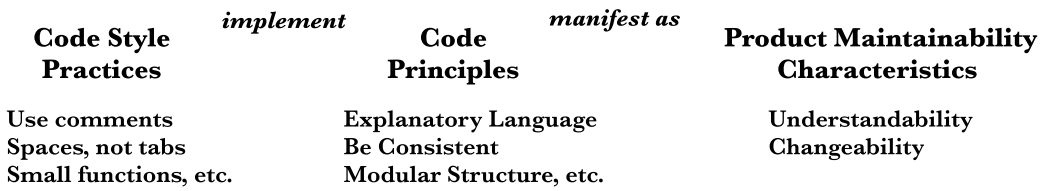


Fig. 1. Software quality abstraction levels.

to understanding and changing code'. Our definition is based on an earlier definition provided by Stegeman et al. [63] (see Section 2.4) and is consistent with the framing of style given by Wiese et al., i.e., concerns making code 'easier for others to read and modify' [73]. The definition means that we do not consider quality characteristics that rely on consulting specifications, for example, correct functionality and robustness and that we exclude aspects of code that affect other product quality characteristics, for example, execution efficiency and security.

We formulated the *Principles* by extracting the code-level advice from established product quality models [8, 23, 32, 43]. We evaluated their fitness for purpose by inviting members of the ACM SIGCSE members mailing list and members of the Australasian Chapter to complete an online questionnaire based on the CSM elements. We refined the *Principles* according to feedback. They are *Explanatory Language*, *Clear Layout*, *Simple Constructs*, *Be Consistent*, *No Unused Content*, *Congruent Implementation*, *Avoid Duplication* and *Modular Structure*. Each is based on a specific *rationale* and describes a core code-level concept. The *Principles* provide a layer between the various rules and conventions and the high-level product quality (sub)characteristics they support, as illustrated in Figure 1. On the left are the Code Style Practices. These are the instructions given to developers in style guides. A Practice describes a concrete implementation that can be associated with an abstract Code Principle (centre). For example, the Practice 'Use comments' is associated with the *Explanatory Language Principle*. Code that adheres to the *Principles* results in a product that is easier to understand and modify, i.e., exhibits the characteristics required for Product Maintainability. We believe that educators can use these *Principles* to explain *why* a style rule exists and to appreciate its contextual nature.

Our long-term goal is to provide a more rigorous means with which to frame discussions on code style and which can be used to form the basis of a validated instrument to support research. The contribution of this article is a model for code style that has been endorsed by the community and provides a resource that educators can use to help students understand *why* a concept is important and to explain the contextual nature of rules adopted by different communities. The CSM model is described in Section 7.

2 Background and Related Work

2.1 Code Style and Maintainability

The ease with which a product can be maintained is an integral aspect of *technical debt* [15]. Technical debt refers to code that does not implement best practices and is often the result of developers taking shortcuts to meet delivery deadlines [2]. A short-term gain is achieved at the expense of long-term sustainability [25]. As the loss of productivity in the software industry due to technical debt is estimated to be between 23 and 45 percent [70], maintainability is a desired *product quality characteristic*. Product quality models group maintainability (sub-)characteristics in various ways [44] but all include notions of *understanding* and *modifying* code [8, 23, 32, 43].

Practitioners state a number of reasons for implementing a guide, including reduction of errors [46, 74], understanding code [31, 54, 68, 74], managing complexity [28] and supporting communication amongst developers by creating a consistent approach [11, 54, 76]. They vary enormously in (a) what they include as style, and (b) specific instructions about how to achieve it. However, most are based on the need to improve understanding [11, 15, 28–30, 34, 42, 46, 54, 58, 68, 74, 76] and many include advice that relates to modifying code [15, 34, 42, 58]. These maps directly to the characteristics for maintainability found in the product models, i.e., following a guide is believed to improve the ease with which the product can be maintained.

2.2 Code Style Guidelines and Standards

Many standards and guidelines exist for code style. Programmers using one of the popular programming languages consult style guides for the language used. For example, the flagship standard for Python programmers is Python's PEP-8 standard [54] and Google has guides for Java [29], C++ [28] and Python [30]. The Apache Foundation based development of one of its Java projects on a set of guidelines and included both general and Java-specific rules [68]. General style guidelines are available on Wikipedia where they are mapped to their language-specific implementation [74]. Many researchers and practitioners have shared their views on style, based on their own experiences. A recent contribution, now a de facto standard, is the book 'Clean Code' by Martin [42]. Several authors have highlighted the importance of adhering to coding standards and best practices, 'to create consistent, high-quality code' [11] and 'to help make error more obvious' by 'acclimating developers to certain patterns' [76]. The Consortium for Information and Software Quality has provided advice on 'Coding Rules for Maintainable Software' [15]. Many academic institutions have produced style guidance for students to follow. For example, Savkovic offers both general and specific advice for students learning Java at the Free University of Bozen-Bolzano [58]. Monash University has provided basic notes on style for Information Technology students [46].

Clearly, there is no lack of material on code style, capturing many perspectives from different sources. However, the range of sources is problematic because educators must trawl through a plethora of information, much of which is aimed at professional developers and unsuitable for use in the classroom, particularly in the early stages. Sources vary in what is included as 'good style' and the advice given is often inconsistent. Strategies of 'comments should be used ... before each method' [68] and 'Good code is self-explanatory' [58] are conflicting, as are 'Having one exit point ... is a good thing' [19] and 'Having multiple exit points simplifies your logic flow' [51]. Some present guidelines as general advice along with illustrative examples, while others provide a set of rules to be followed. Many are language-specific. Some perceive style as being essentially about presentation, for example, variable naming and spacing, some include structural aspects and some include quality characteristics that do not relate to maintainability, for example, performance efficiency and robustness. Schwartz observed that the wide variance among style guides at Google 'makes it difficult to pin down the precise description of what a style guide should cover' [60]. We illustrate this in Table 1 with a representative set of sources. The general/specific nature of the advice given is shown in the first column headed *General/Rules*. The content covered is shown in columns *Struct(ural)* and *Other qualities*.

Style means different things to different people, a notion consistent with the viewpoint of PullRequest, who consider style guides to be 'an opinionated guide of programming conventions, style and best practices' [53].

2.3 Code Style in the Classroom

The teaching of text-based programming generally follows one of two paths. In the first, students are introduced to the constructs of a programming language and consolidate their understanding

Table 1. Variations in Code Style Standards and Guidelines

<i>General/ Rules</i>	<i>Src</i>	<i>Lang.</i>	<i>Struct</i>	<i>Other qualities</i>	<i>Rationale</i>
Rules	[54]	Python	No	None	'... improve the readability of code and make it consistent across ... of Python code'
Gen+Rules	[28]	C++	No	Portability	'Style, also known as readability, is ... the conventions that govern our C++ code'
Rules	[29]	Java	No	None	'avoids giving advice that isn't clearly enforceable'
Rules	[30]	Python	No	None	'BE CONSISTENT'
Rules	[68]	Java	No	None	'Clear, readable, understandable source text eases program evolution, adaptation and maintenance'
Rules	[74]	Several	No	None	'... deal with the visual appearance of source code, with the goal of readability.'
Gen	[34]	Fortran	Yes	None	'... easier to read and understand, and typically smaller and more efficient'
Gen+Rules	[42]	Java	Yes	Functionality Efficiency	'... they are absolutes. They are our school of thought about clean code'.
Gen	[11]	None	No	None	'... consistent, high-quality code'; '... cleaner, more readable and more efficient code with minimal errors'
Gen+Rules	[76]	JavaScript	No	None	'... how your code looks, plain and simple'.
Gen+Rules	[58]	Java	Yes	None	'... people should enjoy reading it'.
Gen+Rules	[15]	Several	Yes	Performance Security	Weakness 407. 'An algorithm ... has an inefficient worst-case computational complexity ... detrimental to system performance and can be triggered by an attacker'

by completing many tasks based on these constructs [1]. This approach is common at tertiary level and in some educational systems at high school [13]. In the second, a project-based approach is taken, where students are encouraged to code in an experimental way, learning the necessary programming techniques as they are needed for solving specific problems. This approach is common at high school, where focus often tends to be on engendering interest and encouraging creativity [48].

Regardless of the approach taken, teachers of text-based programming must help their students understand three core aspects of computer code, i.e., *syntax*, *semantics* and *style*. *Syntax* describes the rules that define how code elements are structured into statements, i.e., the phrases that are permitted in a language [14, 45]. Incorrect syntax prevents a program from compiling and errors must be resolved at compile time. *Semantics* is the formal, mathematically-based study of program meaning [20, 45, 72]. For practical usage, semantics can be broadly considered to address correct functioning, i.e. programs must be free of logical errors. Semantic errors are generally found during testing or programming execution. Resolution requires ensuring the program provides the expected functionality.

Syntax and semantics problems are reasonably well-defined. It is usually clear when such a problem exists and often the computer provides feedback. The third aspect, *style*, is much less well-defined. When discussing writing style for authors, Williams states that complex writing 'may gratuitously complicate simple ideas'. The author presents a set of principles for writing style, for example, 'Clarity' and 'Cohesion' [75]. We see an analogy with the goal of making code simple and clear and placing related elements together, i.e., to the notion of code that is easy to understand and change.

2.4 Code Style Studies

In this Section, we describe some studies on code style and quality in the academic context. We overview the studies in Table 2. The framing terms used by authors, often in the document title,

Table 2. Variations in Code Style Studies

<i>Framing</i>	<i>Scope</i>	<i>Src</i>	<i>Context</i>	<i>Key terms and focus</i>
Quality	Quality	Börstler et al.:2017 [10]	All	Readability, structure
Quality	Style	Stegeman:2016 [63]	Industry	Documentation, presentation, algorithmic, structure
Style	Style	De Ruvo et al.:2018 [21]	Students	Semantic style, control flow
Quality	Style	Keuning et al.:2019 [35]	Teachers	Algorithmic complexity, code clutter
Style	Style	Wiese et al.:2019 [73]	Students	Reading and modifying code, control flow
Quality	Style	Kirk et al.:2020 [38]	Teachers	Need for clearer teaching resources
Quality	Style	Ousterhaut:2018 [49]	Industry	Complexity, dependencies, obscurity
Quality	Style	Keuning et al.:2023 [36]	Education	Documentation, layout, naming, flow, expressions, structure
Quality	Style	Birillo et al.:2023 [7]	Teachers	Assessment templates, naming, layout, unused code, structure
Style	Style	Nurollahian et al.:2024 [47]	Students	Programming patterns, conditionals
Quality	Quality	Tigina et al.:2023 [69]	Students	Linters for quality checking
Quality	Quality	Börstler et al.:2023 [9]	All	Readability, structure, correctness, documentation
Quality	Quality	Řečtáčková et al.:2024 [55]	Students	Code quality defects

are shown in column 1 (*Framing*). Column *Scope* indicates whether the study complies with our definition of style (see Section 1) or includes other qualities. To highlight the lack of distinction between the terms ‘quality’ and ‘style’, we have organised studies according to their overall positioning as ‘quality’ (see Section 2.4.1) or ‘style’ (see Section 2.4.2). To highlight the differences in what was studied, we have included a discussion on the variations in content and terminology (see Section 2.4.3).

2.4.1 Studies Positioned as Quality. An ITiCSE Working Group report on software quality investigated the differences in the views of students, educators and professional developers on ‘code quality’. The authors interviewed twelve students, eleven educators and eleven developers from six countries. Named quality terms included terms describing behaviour (for example, *correctness*) and static structure (for example, *maintainability*). The authors found that ‘there was no common definition of code quality among or within these groups’ [10]. However, for all groups, ‘readability’ was named most frequently and ‘structure’ was the second most cited term.

In a subsequent study by the Working Group, the interview data was analysed with a focus on a more concrete understanding of participant perceptions and how these differed from the more abstract perceptions identified in the earlier work. The authors also explored recommendations for quality improvements [9]. ‘Correctness’ was included as important for quality. ‘Structure’, ‘readability’ and ‘documentation’ were cited by professional developers as being important for improvement. These were viewed by participants as a prerequisite for ‘comprehensibility’ and ‘maintainability’.

Stegeman et al. analysed professional code quality standards and interviewed software professionals to identify a set of criteria to form the basis of a marking rubric for introductory programming courses [63, 65]. The authors considered code quality to be ‘an aspect of software quality that can be determined just by looking at the source code, i.e., without any form of testing, or checking against specification’ [64]. They stated that this identification ‘precludes concepts like efficiency, portability and conformance to a specification’ [64]. Code quality aspects were categorised into ten *quality criteria*, relating to *documentation* (naming, headers, comments), *presentation* (layout, formatting), *algorithmic* (flow, idiom, expressions) and *structure* (decomposition, modularisation). Our observation is that, regardless of the authors’ intent, both naming (‘code quality’) and definition suggest broader quality aspects. For example, a poor search routine can be identified by simply looking at the code and this affects efficiency.

Keuning et al. investigated teachers’ focus on code quality when teaching novice programmers and the kinds of quality issues they observed [35]. They asked thirty teachers about the feedback

they would give when presented with low-quality student code. The feedback from different teachers varied widely although many had similar views on reducing algorithmic complexity and code clutter.

The authors also carried out a systematic mapping study into code quality in education. The authors pointed out that ‘Code quality is a term without a clear meaning and with various interpretations’ [36]. They focused the study on aspects of code that contribute to maintainability, and based analysis on the rubric created by Stegeman et al. The authors found a growth in publications about code quality over the years, a variety of languages, and the prevalence, and growth in the number, of papers describing tools.

Kirk et al. explored the disconnect between high school student assessment expectations with respect to quality and three approaches to computing education. They found a variation in the quality-related content of the assessment systems with a ‘significant variation in the perceived importance of introducing beginner programmers to the notions and rationales relating to code quality’ [38]. Findings from a subsequent study indicated that high school teachers struggle with teaching code quality and had a need for clearer, more consistent resources [39].

Birillo et al. developed an algorithm for exposing code quality issues in code templates. Templates are commonly used for online assessment but isolating errors in the template is difficult. For example, an unused variable, intended for the student to use, will incorrectly be picked up as an error [7]. The authors applied the algorithm to 415 Java tasks in a template dataset and found that 14.7% contained at least one error. The quality aspects checked for in their study were not specified in full but included formatting, naming, unused code and code structure.

Tigina et al. analysed over 2.3 million Java and Python submissions from online courses to evaluate the quality of students’ code and to find out how students approach fixing errors [69]. The authors used the Hyperstyle tool to identify code quality issues. This tool runs a set of linters for Java and Python for error detection. Issues checked included aspects of naming, expression complexity, unused code and missing code. They found that unused code (local variables and imports) and incorrect use of null were most prevalent in Java code. For Python code, a lack of adherence to Python structure and naming conventions was most prevalent.

In the context of professional developers, Ousterhout believed the key consideration when designing software is *complexity*. He stated the term as ‘... anything related to the structure of a software system that makes it hard to understand and modify the subsystem’ [49]. This maps directly to our notion of style. The author suggested the two main causes of complexity are *dependencies* and *obscurity* as these lead to seemingly simple changes that require many code modifications and involve a high cognitive load.

Řechtáčková et al. proposed a framework to organise the code quality defects made by programming students [55]. The authors catalogued the 80 defects most commonly found in student code, focusing on those made by novice students. To determine the severity of the defects, 30 representative defects from the catalogue were presented to 72 educators and classified into a set of ‘defect types’, for example, ‘unused’, ‘unsuited construct’, ‘simplifiable’ and ‘poor name’. Each type was mapped to a higher level code description (‘reason to fix’), for example, ‘tidiness’, ‘explicitness’, ‘conciseness’ and ‘maintainability’. The mappings are not backed by a *rationale* that explains why and how fixing the defect will achieve the goal, for example, why does simplifying code make it more precise? We address this gap in our study by basing *Principles* on an underlying *Rationale*.

2.4.2 Studies Positioned as Style. De Ruvo et al. examined code samples from over 900 students to explore the kinds of problems students have with choosing the best style for achieving the desired functionality, i.e., with ‘semantic style’. The authors identified sixteen style indicators, each of which indicates a potential lack of programmer knowledge. Many of the indicators are related to faulty

‘if-else’ constructs. Some were present in code written by final-year students. The authors suggested that the style indicators may be useful for identifying students’ misunderstanding of programming concepts as they indicate a ‘lack of knowledge in one or more aspects of programming’ [21].

Wiese et al. view good style as making code ‘easier for others to read and modify’ [73]. They reported findings from the pilot of a replication study to examine the issues that novice programmers have with code style. The authors focused on control flow, ‘one element of style where experts expect particular structure, such as conjoining conditions rather than nesting if statements’. They invited 231 undergraduate students to answer questions relating to seven control flow patterns and using code from both ‘experts’ and novices. They found that novices found expert-style code less easy to comprehend than novice code for some style patterns and did not always prefer to read code written with good style. They concluded that students ‘do not acquire the skills of recognition, usage, comprehension ... at the same time’ [73]. We observe that understanding depends upon the characteristics of the reader and this highlights the contextual nature of what is perceived as ‘good style’.

Nurollahian et al. also focused on control flow in an investigation into students’ use of code structure with increasing experience [47]. The study was framed in terms of programming patterns, with a focus on returning Booleans and the use of repeated code within conditionals. The authors conducted a comparative study involving 354 students, 149 from a CS1 course and 205 from a CS2 courses. They found that CS2 students were better at identifying the higher quality patterns and a smaller number were better at assessing code readability.

2.4.3 Study Focus and Terminology. The variation in framing and findings illustrated in Table 2 indicates a considerable lack of agreement with respect to terminology. Börstler et al. asked teachers, students and professionals to define and describe features of code quality and found that answers ‘tended to be a mix of abstract characteristics and indicators with unclear definitions of the terms’ [10]. Descriptions for ‘Readability’ included ‘readable, no useless code, brevity/ conciseness, formatting/ layout, style, indentation, naming convention’ [10]. The analysis of professional handbooks carried out by Stegeman et al. to define a model for understandability found 401 items. Many of the sub-categories established after grouping contained multiple terms, for example, 108 terms mapped to ‘naming’ and 32 to ‘minimalist comments’. In an early study to investigate types of programming knowledge, Soloway and Ehrich describe the ‘rules of programming discourse which capture the conventions in programming’ [62]. ‘Programming discourse’ would appear to be synonymous with current descriptions of code style. The implication is a plethora of terms that overlap in meaning.

In addition to inconsistencies in the overall framing, studies vary according to the features studied and the terminology used. Researchers have chosen a small set of features that represent ‘quality’ or ‘style’. Wiese et al. [73] pointed out that ‘Control flow is one element of style’ and focused on conditionals and function returns. The authors coined the phrase ‘code patterns’ to describe specific implementations. In a later paper, the control flow aspects of code were described as ‘code structure’, a term often applied to modularity, and the term ‘programming anti-patterns’ applied to describe unhelpful implementations [47]. De Ruvo et al. studied ‘semantic style indicators’ to describe ‘features that an experienced programmer would typically avoid’ [21]. Indicators manifested as specific implementations of statements and control flow features, for example, conditionals.

This lack of consistency is an example of the vague use of terminology reported in the **Software Engineering (SE)** literature [16, 37, 61]. We believe that this situation is extremely damaging for the CSE community because we cannot rigorously research what we cannot define. We intend the CSM to act as a starting point for discussion and feedback with the goal of achieving consistency within the teaching community.

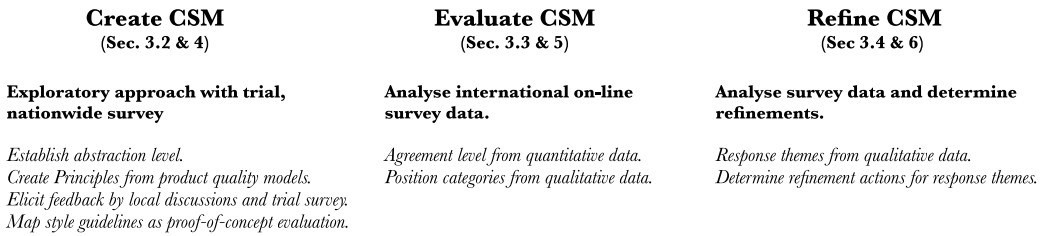


Fig. 2. Research process overview.

3 Research Approach

Routio described three kinds of research (a) there is no model to use as a starting point (exploratory research), (b) an existing model is being expanded or refined and (c) hypotheses based on an established model are being tested. Exploratory research is appropriate for a ‘phenomenological pursuit into deep understanding’ and where there is ‘distrust on earlier descriptions’. The researcher begins with a ‘preliminary notion’ of the object of study. During the study the ‘provisional concepts ... gradually gain precision’ until a suitable conceptualisation is achieved. Routio suggests that the journey may involve some ‘creative innovation’ [57]. In this research, we applied a) (for initial model-building) and b) (when refining the model as a result of community feedback). We will carry out formal hypothesis testing on code style (c) when confident that the CSM is ready for use as an established research model. We overview our approach in Figure 2.

For initial model building, we adopted a *pragmatic* worldview as being consistent with the nature of exploratory research. Other popular paradigms are *Postpositivism*, (which generally concerns causation and hypothesis-testing), *Constructivism* (which views knowledge as being constructed by individuals as a consequence of their experiences) and *Transformative* (which focuses on political effects on minority groups). In the *Pragmatic* paradigm, researchers equate truth with ‘what works’ and use any means available to understand the problem) [17]. The pragmatist considers theories as ‘the products of a consensual process ... to be judged for their utility’ [24]. This paradigm involves a focus on solutions to problems and supports the use of all available approaches to better understand the problem space [17].

Stol and Fitzgerald classify SE research strategies into eight archetypes. These are based on the extent to which the researcher disturbs the research setting (*obtrusiveness*) and to which findings can be generalised (*generalisability*). For example, a ‘Field Study’ tends to yield findings that are specific to the environment (low generalisability) but the level of obtrusiveness may be low (for example, if the researcher simply observes) or high (for example, if participants are divided into treatment groups) [67]. These ideas informed our analysis of suitable strategies for collecting data for model refinement and caused us to implement a survey (high generalisability, low obtrusiveness) rather than a Delphi study (lower in generalisability, higher in obtrusiveness).

3.1 Research Question (RQ)

The RQ for this research was

RQ: How might we reconcile the vagueness and inconsistencies around discussions on code style?

Our study is situated in an area that is not well understood. Our approach was to build a model of code style (CSM) that would reconcile the inconsistencies. Our aim was not to produce a model that was theoretically ‘correct’ as we do not believe this is possible when discussing code style. Rather we wanted to provide a useful and generally agreed basis for discussion, i.e., we took a

pragmatic worldview [17]. We engaged in a study with three phases, model building, evaluation and refinement [57]. In order to give us confidence in the validity of the model, we created **Research Objectives (ROs)** for each phase.

For model building, we adopted an exploratory approach [57]. For evaluation and refinement, we understood that feedback from practising computer science educators was essential. This meant that we rejected approaches that involved literature-based analysis. Possible approaches were to conduct a Delphi study or to implement a survey questionnaire. A Delphi study would provide deeper insights into participants' viewpoints on style. However, we believed that, given the highly subjective nature of style preferences and the tenacity with which beliefs are held, we could not be confident that iterations would lead to consensus. In addition, the sample would be smaller (low *generalisability*) and a burden placed on participants with respect to an expectation to attend meetings (high *obtrusiveness*). The relatively small number of participants would also reduce the possibility of hearing an unusual but insightful observation. A survey questionnaire, on the other hand, would lead to more generalisable findings and we hoped that its low obtrusiveness might encourage participation and elicit interesting viewpoints on style.

3.2 Creating the CSM

As the model-building phase involved some 'creative innovation' [57], it was necessary that we report how the model was built in order that others could understand and evaluate it. To give us confidence in the validity of the model, we based the model on the literature and consulted with experts.

The RO for this phase was:

RO1 Clearly articulate the process of model-building.

Model building comprised the following steps (see [41]):

- (1) Establish a suitable abstraction level.
- (2) Extract from the product quality models the code-level descriptions for understanding and modifying code and from these create a set of *Principles*.
- (3) Elicit informal feedback on the *Principles*.
- (4) Determine the feasibility of our approach by applying the *Principles* to guidelines from a range of sources to demonstrate proof of concept.

To identify a suitable abstraction level (step 1), we considered our aim, i.e., to have a mechanism for mapping low-level instructions from style guides to high-level product quality characteristics. The abstraction level would determine which code-level descriptions we would extract from the product quality models to use as a basis for the *Principles* (step 2). Possibilities ranged from more abstract (for example, 'readability') to more concrete (for example, 'naming', 'formatting'). The former appears as sub-characteristics in product quality models and the latter as terms in style guides. As we wanted to map between the two, we looked for descriptions that were 'in-between', for example, 'simple' and 'concise'.

For step 2, we extracted the product quality (sub-)characteristics that relate to understanding and modifying code from the ISO/IEC, Boehm, McCall and Dromey quality models [8, 23, 32, 43]. For example, the Boehm et al. model has 'understandability' as a sub-characteristic of 'maintainability' [8]. For each, we examined the (sub-)characteristic descriptions provided in the model to extract terms that described code properties.

Steps 1–2 involved a process of weekly discussion and negotiation among the authors. For step 3, we held informal, in-person discussions, implemented a trial and nationwide survey and consulted with a small number of known practitioners. Discussions involved the members of the computing

education research group *CELT*, a group comprising 34 active members from the computer science and SE departments at the University of Auckland. A presentation was delivered at the weekly group meeting, followed by a discussion on the *Principles*. Members were invited to email the authors with feedback not provided during the meeting.

Feedback was used to inform a survey questionnaire which was disseminated as a trial online survey to both the *CELT* group and to *Software Innovation New Zealand*, a nationwide group of 34 SE educators and researchers. Questions included both Likert-scale and open questions of the form ‘please explain’. We asked recipients if they agreed with the definition of code style. For each *Principle*, we asked the recipient how well they *understood* the *Principle* and its application to code, whether the *Principle* supported *understanding* and *changing* code and if they could use the *Principle* to support the *teaching* of code style. Illustrative examples were provided for each *Principle*. We also asked if they had anything further to add about the research.

For step 4, we carried out a proof-of-concept evaluation of the *Principles* by mapping guidelines from a representative set of sources [11, 28, 54, 65] and heuristics from the book ‘Clean Code’ [42] to check that each rule, guideline and/or heuristic could be classified under a *Principle*. Success in mapping concrete rules from guidelines to *Principles* extracted from product quality models would provide us with confidence that the *Principles* are fit-for-purpose for linking specific rules (Code Style Practices in Figure 1) to high-level product quality characteristics (Product Maintainability Characteristics in Figure 1). As we wanted to take a language-independent perspective, we mapped only guidelines that were inherently non-language-specific. We did not include guidelines appropriate for more experienced programmers, for example, concerning the use of namespaces, because our target demographic is educators. We chose the book ‘Clean Code’ because it is a de-facto standard for programmers.

3.3 Evaluating the CSM

Evaluation involved testing that our conceptualisation represented ‘a starting point (e.g. a framework) that identifies aspects of a topic’ [66] for the CSE community. The RO for this phase was:

RO2 Evaluate the level of community support for the Principles.

As we aimed for findings to be highly generalisable, we planned to include CS educators who represented a wide range of contexts. We implemented an online survey and invited the members of the ACM SIGCSE members mailing list and of the Australasian Chapter to participate. We did not qualify for participation in any way, i.e., all computing educators on these lists were invited to participate. The SIGCSE membership represents over 2,500 members from 66 countries [4], i.e., we distributed to a large, international population of CS educators. This is appropriate for research in a *neutral setting*, with low *obtrusiveness* [67]. We hoped that the neutrality and lack of obtrusiveness would result in a high response rate and that the number and depth of responses would give us confidence in the generalisability of findings with respect to the CSE community. Ethics approval was granted by the University Ethics Committee.

3.3.1 Questionnaire. The questionnaire was based on the outcomes of the trial described in Section 3.2. Demographic information included programming experience both as an educator and as a practitioner. We included Likert-scale questions to elicit viewpoints on the usefulness of the CSM when used for programming and for teaching. We implemented a five-option scale as this would allow for a neutral response, provide a sufficient range of response options and fit more easily on mobile screens than a seven-option scale. We illustrate below for the Code Style *Definition* and a placeholder *Principle*.

A1a *To what extent do you agree with the description of code style as ‘aspects of software quality, related to understanding and changing code, that can be determined by looking at the source code’?*

B1 *To what extent do you agree with these statements about the placeholder principle?*

B1.1a *I understand the description of the principle and how it applies to code.*

B1.1b *Code developed by following the principle would be easier for me to understand and/or change.*

B1.1c *I could use the principle to support teaching style concepts.*

We included an open question for the *Definition* and for each *Principle*. Open questions for the *Principles* were phrased as *If you have anything to add, please do so in the text area below*. Our reasoning was that we did not want to discourage respondents, i.e., we wanted to make it clear that a response was required only if the respondent felt a need to explain. We included an open question at the end to catch more general thoughts.

3.3.2 Analysis. We applied standard descriptive statistics to analyse the quantitative data from the Likert-scale questions. This would provide us with a sense of the overall level of agreement for the *Definition* and *Principles*. For each *Principle*, we then classified responses according to the participant’s overall position. The aim was to establish categories that provided insight into the participant’s underlying position on the *Principle*.

To do this, we applied a strategy of inductive thematic analysis based on open coding to the open comments [12] to create categories that captured participants’ thoughts on the *Principles* and extended and modified these according to the Likert responses. For example, if a respondent voiced disagreement about some aspect of a *Principle*, we categorised as ‘Some reservations’ if the Likert values were positive and as ‘Disagree with framing’ if the values were negative and/or we judged the aspect to be a fundamental one. The outcome was a set of strawman categories to use a starting point for open comment analysis. The process involved one author setting up a candidate set of categories based on data for the first *Principle* and all authors critiquing and modifying these while analysing the remaining data during weekly meetings.

3.4 Refining the CSM

The ROs for this phase were:

RO3.1 *Justify the changes made to the CSM as a result of recommendations made by the CSE community.*

RO3.2 *Identify the issues that arise when attempting to develop a community-wide CSM.*

The analysis during the evaluation process resulted in a set of categories that represented participants’ *overall* attitudes towards the *Principles* (see Section 3.3.2). We used the Framework Method [26, 52] and seeded the Framework with these categories. The Framework Method is suitable when ROs have been set in advance and categories have been determined based on these. We coded the open comments according to these categories. This is a *referential* category structure, as the intention of this structure was to record textual references [56].

The process involved one author classifying the data and all authors checking that all code excerpts that might indicate a change to the CSM had been correctly classified. The relevance of inter-rater reliability is questionable for qualitative analysis [3, 52] and so we implemented group discussion and negotiation at all stages throughout the analysis. This is consistent with the notion that qualitative analysis is ‘better conducted as a group activity’ [3].

We identified the open comment categories that would indicate a possible CSM refinement. We included categories that were associated with positive Likert values that represented an underlying

positive attitude, for example ‘Some reservations’, because we did not want to miss any suggestions for minor improvement or opportunities for insights.

4 Creating the CSM

In this Section, we overview the steps taken when creating the CSM (see Section 3.2). A full description is available in a previous study by Kirk et al. [41].

We based our *Definition* on the code quality definition provided by Stegeman et al., i.e., ‘*an aspect of software quality that can be determined just by looking at the source code, i.e., without any form of testing, or checking against a specification*’ [64]. Although the intention was to exclude quality characteristics such as efficiency (see Section 2.4), the Stegeman et al. definition as stated does not make this clear. We extended the *Definition* to ‘*aspects of software quality that can be determined by looking at the source code, constrained to understanding and changing code*’.

The *Definition* restricts our interest to situations where existing code is reorganised to effect improvements and excludes situations that involve additional functionality, for example, extra code to handle security requirements or missing conditions in an ‘if’ block [15]. These may cause errors in execution, but are not style errors by our definition. The definition also means that we exclude aspects that affect execution efficiency. An inefficient search algorithm is out of scope, as is using a data structure that is inefficient, for example, replacing a two-dimensional list in Python with a dictionary.

4.1 Establishing Abstraction Levels

Our choice of abstraction level was informed by the works of McCall et al. [43] and Dromey [22]. During product quality model creation, McCall et al. observed that the candidate terms ‘modularity’ and ‘complexity’ described the *software* rather than the *product* and named these *criteria*. Each criterion may relate to multiple product quality characteristics and can be implemented in various ways in the source code [43]. Dromey believed that the product quality standards did ‘not go nearly far enough to support building quality into software’ and that quality must be built into the code [22]. He explored the notion of *quality carrying properties* that could be associated with code elements and would be linked to product quality attributes, for example, ‘maintainability’. Examples of quality carrying properties were *Self-Descriptive* and *Consistent* [22].

Although neither of these works focused on modelling at the software level, they caused us to realise that an abstraction at the level of the software would form a bridge between code implementation guidelines and product quality characteristics.

In Figure 1, we show the abstraction levels adopted. On the left are the code-level style practices. When implemented, these support a number of higher-level code properties (*Principles*), for example, ‘Consistent Design’ and ‘Modular Structure’. These properties manifest as the achievement of *product* quality characteristics. For example, using small functions characterises modularity and modular code is more maintainable.

4.2 Identifying the Principles

In Table 3, we show an illustrative set of the terms found in the product quality models (see Section 3.2) along with the product quality characteristics affected (column 2). In column 3, we include a candidate name for a code style *Principle* (column 3).

To establish candidate names for the *Principles*, we considered the meanings given in the quality model descriptions. For model terms that were associated with a single meaning, the term was carried over to column 3 as the candidate name. However, the description for ‘Adjustable’ referred to hard-coded values and to avoid confusion with the notion of ‘easy to change’ with implications of modularity, we renamed to ‘Configurable’. Terms that occurred in multiple product quality

Table 3. Illustrative Product Quality Terms with Mappings to the Candidate Names for Code Style Principles

<i>Model term</i>	<i>Quality characteristic</i>	<i>Principle</i>
Simple	Maintainability [43]	Simple
Self-descriptive	Understandability [8], Maintainability [43], Flexibility [43]	Self-descriptive
Modular	Flexibility [43], Modularity [32]	Modular
Non-redundant	Maintainability [22]	Non-redundant
Adjustable	Maintainability [22]	Configurable
Concise	Understandability [8], Maintainability [43]	Non-redundant, Simple

models sometimes had different meanings. For example, meanings for the term ‘Concise’ were *excessive information is not present* [8], *implementation of a function with a minimum amount of code* [43]. ‘No excessive information’ is synonymous with ‘Non-redundant’. ‘Minimum code’ generally means writing code that is structured in a way that reduces clutter but this might be interpreted as advocating condense, obtuse, overly complex constructs. Both are on the spectrum of ‘Simple’. We did not include ‘Concise’ as a candidate name as this was covered by ‘Non-redundant’ and ‘Simple’.

After renaming and removing the principles that represent duplicate concepts, eight principles remained. These represented the initial candidate names that would form a basis for discussions and feedback. *Principles* were:

Self-descriptive Intent and meaning of elements can be inferred from reading the code and documentation.

Clear Content is laid out such as to make the intended program structure easy to discern.

Simple Constructs implemented with minimum complexity.

Consistent Uniform notation and design techniques throughout.

Non-redundant No superfluous content that adds no value, e.g., contiguous ‘break’ statements, repeated or commented-out code.

Appropriate Implementation is congruent with the problem to be solved, i.e., suitable abstractions.

Configurable Code can be easily configured e.g. does not contain hard-coded values, load strings at runtime.

Modular Code is written with high cohesion and low coupling.

4.3 Eliciting Feedback on the Principles

Feedback from the informal discussions and trial survey (see Section 3.2) exposed some inconsistencies, provided some suggestions and highlighted points for consideration. For example, it was suggested that it was not obvious what a single-word term such as ‘Clear’ meant and that replacing this with a two-word term such as ‘Clear Layout’ would help with understanding meaning. One colleague referred to the work of Parrish [50] and suggested that descriptions could be improved by separating presentation into what the *Principle is* and *how to achieve* it. Informal discussions highlighted the need for tradeoffs. When discussing the meanings for the ‘Configurable’ principle (see Section 4.2), we realised that loading strings at run-time as opposed to embedding in the source was covered by ‘Modular’. We concluded that the underlying issue with hard-coded values was one of *duplication*, i.e., values repeated multiple times where one might be missed when the code is changed. We understood that modularity and duplication are essentially different as the first involves putting things in the right place and the second involves ensuring a single occurrence. This resulted in a slight rewording of the survey questions and a further renaming of the *Principles* (see [41]).

Table 4. Illustrative Mapping of the Style Practices Sourced from the Literature for the Principle ‘Explanatory Language’

Principle	Practices
Explanatory Language	headers use headers [63] (<i>module</i>), accurately summarise role and usage of parts [28, 63] (<i>module</i>), concise [63] (<i>all</i>); functions should have preceding headers unless simple and obvious [28] (<i>routine</i>), function header describes what it does and how to use [28] (<i>routine</i>); comments explain code and potential problems [11, 63] (<i>all</i>), use correct English [54, 63] (<i>all</i>), use complete sentences [54] (<i>all</i>), understandable to your audience [54] (<i>all</i>), write for your audience [28] (<i>all</i>), must be up-to-date [54] (<i>all</i>), comment global variables [28] (<i>data stores</i>), use inline comments sparingly [54] (<i>all</i>); names describe intent and usage [28, 54, 63] (<i>all</i>), correctly spelled [63] (<i>all</i>), distinctive [63] (<i>module, routine, data store</i>), use English words [54] (<i>module, routine, data store</i>), never use ‘I’, ‘O’, ‘T’ [54] (<i>data store</i>), not abbreviated [63] (<i>routine, data store</i>), appropriate to scope of visibility—universally known abbreviations e.g. ‘i’, ‘ok’ [28] (<i>data store</i>), no unnamed constants [63], constrain length of variable names (<i>data store</i>); code comments to describe code function [11] (<i>all</i>), best code is self-documenting, self-document your code [11, 28, 63] (<i>all</i>)

The resulting *Principles* after the trial survey were *Explanatory Language*, *Clear Layout*, *Simple Constructs*, *Consistent Design*, *Non-redundant Content*, *Appropriate Implementation*, *Avoid Repetition* and *Modular Structure*. The CSM comprised a *Code Style Definition* and the eight abstract *Code Style Principles*. Each *Principle* includes a *Rationale* and some *Guidelines* for implementation. An example is shown below for the *Explanatory Language Principle*. A full description is available in a previous study by Kirk et al. [41].

Explanatory Language

Principle The intent and meaning of code is explicit.

Rationale Being explicit in describing the purpose of the code elements helps us understand the author’s intention, thus improving understandability.

Guidelines

Comments and headers should describe intent and meaning and be grammatically correct and written in correct English.

Names of elements (variables, functions, etc.) should be descriptive and contextually appropriate.

Embedded literals should be extracted as named constants.

4.4 Mapping the Principles

In Table 4, we illustrate the mapping from the practices described in style guidelines to the *Explanatory Language Principle*. The full mapping can be found in our previous work [41]. Practices from the various sources are shown in the second column (Practices). For example, the practice ‘no unnamed constants’, from the Stegeman et al. rubric [63], belongs to the *Explanatory Language Principle*. In some cases, the statement of the practice was non-specific. For example, ‘Appropriate choice of control structures’ might refer to *Appropriate Implementation* (appropriate abstraction) or *Simple Constructs* (implemented with minimum complexity). In these cases, we made a judgement call. Some of the practices are general and relate to different kinds of element. For example, ‘Names’ relate to variables, constants, functions, etc. As we wanted to provide specific advice for educators, we appended each entry in Table 4 with the code elements to which it applies, organised as:

Module The files that make up one high-level piece of the program, or the program itself for a small program.

Routine Functions, procedures, methods, sub-routines.

Data store Variables, constants, lists, arrays, records, etc.

Table 5. Illustrative Mapping of Clean Code Heuristics [42] for the Principle ‘Explanatory Language’

<i>Principle</i>	<i>Heuristics</i>
Explanatory Language	C4:Poorly Written Comment; G16:Obscured Intent; G19:Use Explanatory Variables; G20:Function Names Should Say What They Do; Replace Magic Numbers with Named Constants; N1:Choose Descriptive Names; N2:Choose Names at the Appropriate Level of Abstraction; N4:Unambiguous Names; N5:Use Long Names for Long Scopes; N7:Names Should Describe Side-Effects

Table 6. Mapping of Clean Code Heuristics [42] That Did not Map to Software Code Principles

<i>Category</i>	<i>Heuristics</i>
Functional correctness	G3:Incorrect Behaviour at the Boundaries; G4:Overridden Safeties; T1:Insufficient Tests; T5:Test Boundary Conditions; T6:Exhaustively Test Near Bugs
Performance efficiency	T9:Tests Should Be Fast
Process	T2:Use a Coverage Tool; T3:Don’t Skip Trivial Tests; T4:An Ignored Test is a Question about an Ambiguity
Meta-level	G21:Understand the Algorithm; T7:Patterns of Failure are Revealing

Control structure Loops and conditionals.

Expression Assignments, Boolean/logical expressions.

All of the above.

For example, ‘data types appropriate’ for the ‘Appropriate Implementation’ Principle applies to the ‘Datastore’ category.

In Table 5, we illustrate the mapping of the heuristics described by Martin [42] to the *Explanatory Language Principle*. We found that some heuristics did not comply with our definition of code style. We show these in Table 6. They include items that relate to functional correctness (‘include tests for boundary conditions’), performance efficiency (‘tests should be fast’), process (‘run all tests’) and other kinds of advice (‘understand the algorithm’). This reflects the fact that the advice given by Martin is not limited to our notion of code style, but covers ‘good code’ in a more general way.

5 Evaluating the CSM

We evaluated the CSM by analysing the Likert responses from the survey questionnaire and eliciting the participants’ underlying positions on the *Principles* (see Section 3.3).

There were 84 responses. All 84 responded to the question on style *Definition*. Fifty-two completed the question on the first *Principle*, 51 on the next five *Principles*, and 50 on the last two. These 50 also completed the demographic information (see Figure 3). Forty of the 50 respondents had more than 10 years of programming experience as an educator and 29 had more than 10 years of experience as a practitioner. The indication is that feedback was based on significant practical programming experience.

5.1 Quantitative Data

In Figure 4, we show the responses for question *A1a*, the Likert-scale question for Code Style *Definition* (see Section 3.3.1). Of the 84 responses, 65 (77%) were positive, i.e., ‘Agree’ or ‘Strongly agree’ and 14 (17%) were neutral, i.e., ‘Neither agree or disagree’. These suggest that the community perspectives on what style is are generally aligned with ours.

In Figure 5, we show the Likert responses for the eight *Principles*. The *Principle* and number of responses are shown in the title of each graph. Responses are highly weighted to the positive end of the scale. In Table 7 we show the percentage values for the ‘Strongly agree’ and ‘Agree’ categories only. The values in column two relate to the three sub-questions for the *Principle* (see Section 3.3.1).

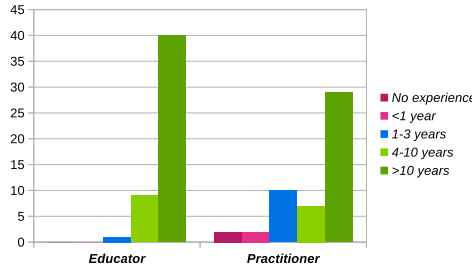


Fig. 3. Participant experience (palette by [71]).

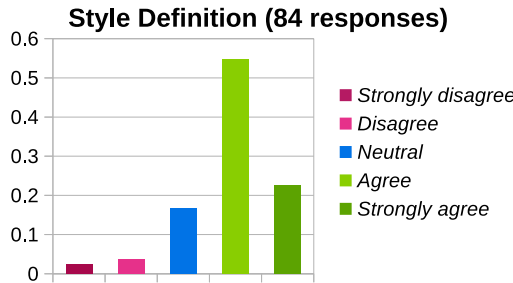


Fig. 4. Likert-scale responses for code style definition.

Table 7. Principles Positive (Agree and Strongly Agree) Responses are Shown as a Percentage

Principle	Percentages		
	a	b	c
Explanatory Language	98	88	90
Clear Layout	94	92	88
Simple Constructs	94	84	76
Consistent Design	92	78	78
Non-redundant Content	94	86	92
Appropriate Implementation	76	74	66
Avoid Repetition	100	86	90
Modular Structure	96	92	80

The percentage values relate to the sub-questions: (a) *I understand the description of the principle and how it applies to code.* (b) *Code developed by following the principle would be easier for me to understand and/or change.* (c) *I could use the principle to support teaching style concepts.*

The Likert responses in Figure 5 and the percentage of responses indicating agreement in Table 7 are consistently high for all Principles.

5.2 Qualitative Data

Of the 50 participants who fully completed all parts of the questionnaire, 44 provided open-question comments. Of the 34 who only partially completed, 10 provided open-question comments.

The categories we identified as representing the respondent’s underlying position on the Principles (see Section 3.3.2) were:

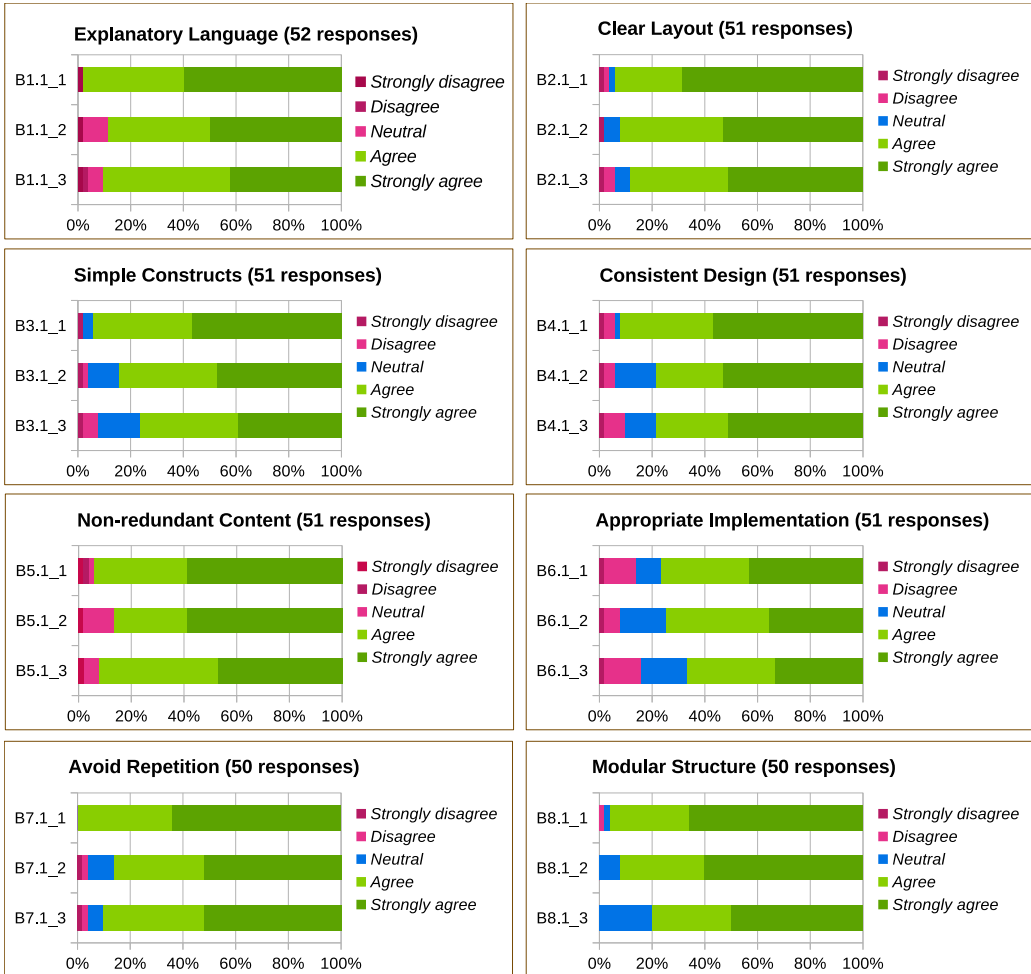


Fig. 5. Likert-scale responses for code style principles.

New insights The respondent noted a beneficial feature of the *Definition* or *Principle*. For example, one respondent liked the focus on *why* style matters (*What I really like about your definition here is this much more focused approach of what the actual 'objective' of code style is.*)

General agreement The respondent indicated an agreement with the *Principle* without any new observations being voiced (*Code style can be assessed by examining the source code and does not include correctness or efficiency, which would be assessed in other ways (e.g., testing and experiments).*)

Modification suggestions The respondent agreed in principle but suggested a modification, for example, to terminology (*I don't think that 'looking' is the best verb here. 'Examining' or something conveying a range of tasks from reading to reflecting is closer to what I would mean.*)

Could extend The respondent agreed in principle but believed something should be added, for example, the need to address context within the definition (*I guess what I feel is missing, is wanting to go towards the 'personalised' aspect of code style.*)

Table 8. Distribution of Response Categories in Open-Question Comments by Principle

Style element	New insights	General agreement	Modification suggestions	Could extend	Misaligned interpretation	Disagree framing	Teaching thoughts	Agree/Disagree/Total
Definition	3	15	16	6	3	6	0	40/6/49
Explanatory	1	3	16	7	5	7	4	27/7/43
Clear	0	8	4	2	1	1	3	14/1/19
Simple	0	7	12	3	7	7	1	22/7/39
Consistent	0	2	11	1	4	5	1	18/5/24
Non-redundant	0	5	8	0	2	5	0	13/5/20
Appropriate	0	3	11	3	6	11	1	17/11/35
Repetition	1	4	12	1	5	2	0	18/2/25
Modular	0	2	12	2	0	5	2	16/5/23
Total	5	49	102	2	33	49	12	158/49/275

Misaligned interpretation The respondent appears to have interpreted the CSM elements in a way we did not intend. We agree with the comment *There are other aspects of software quality that can be determined by looking at the source code that I wouldn't characterize as 'code style'*.

Disagree with framing The respondent indicated a lack of agreement with a fundamental aspect of the CSM. For example, several respondents believe that style includes quality characteristics other than maintainability (... *but would also add run time efficiency*).

Teaching thoughts The respondent shared some thoughts on how to teach an aspect of style (*Much of this can be done utilising tools within packages (such as Javadocs)*).

When categorising, we first split complex open responses into discrete comments. For each comment, we then took into account the Likert rating. For example, many commented on the contextual dependence of the *Principles*. If the comment was clearly intended as a criticism of the *Principle* and associated with negative Likert ratings, we categorised as 'Disagree with framing'. If included in a longer comment as an 'interesting observation' and associated with positive Likert ratings, we categorised as 'Modification suggestions'. When the intended meaning of the comment was unclear, for example, whether 'efficiency' meant 'run-time efficiency' or 'code efficiency', we made the best guess.

In Table 8, we summarise the distribution of comments with respect to the above categories. Note that the number of comments for each row do not always sum to the corresponding number of participant responses as some responses covered multiple categories.

The categories *New insights*, *General agreement*, *Modification suggestions* and *Could extend* represent a positive perspective and the category *Disagree with framing* represents a negative one. The remaining categories, *Teaching thoughts* and *Misaligned interpretation* are agnostic, in that they represent implementation ideas and misunderstandings. We observe in Table 8 a relatively large number of comments suggesting minor modifications.

6 Refining the CSM

Refinement was based on an analysis of the open responses in the categories that we identified as indicating a possible change to the CSM. The candidate categories were *Modification suggestions*, *Could extend*, *Misaligned interpretation* and *Disagree with framing*.

To give us confidence that we had good insight into any reasons for disagreement, we wanted to ensure that participants who had issues with the CSM, i.e., gave a negative or neutral response in the Likert questions, also provided us with a reason for disagreement as an open response. A high percentage of negative viewpoints accompanied by reasons would give us confidence that we had good insight into any reasons for disagreement.

Table 9. Count of Responses, Comments and Negative/Neutral Responses (Total and with Comments) for Each Question

Question	Responses	Comments	Neg and neutral responses/with comment
A1	84	37	19/15
B1	52	26	9/8
B2	51	16	9/6
B3	51	32	14/13
B4	51	20	13/10
B5	51	20	7/6
B6	51	28	18/16
B7	50	20	7/6
B8	50	21	10/8

We calculated the percentage of negative and neutral Likert responses that included open comments. For the open questions B1-B8, we counted a response as negative or neutral if one or more of the three Likert-scale sub-questions were negative or neutral. In Table 9, we show the total number of responses, responses accompanied by comments and the negative and neutral responses with comments. For example, question A1 had 84 responses, 37 of these accompanied by an open comment. Of the 37, 19 were negative or neutral and only 4 of these did not have a comment. In total, comments were provided for 88 of the 106 negative or neutral responses i.e. 83%. The relatively small number of missing explanations means that we had a comprehensive picture of the reasons for any lack of agreement.

6.1 Response Themes

When considering how we would respond to comments in the candidate categories, we realised that many of the comments required a similar response. For example, comments that pointed out the general or abstract nature of a *Principle* would not result in a refinement to the *Principle* because the abstraction level has been chosen deliberately. We identified the following response themes. Themes were ‘*Principle-agnostic*’ and dictated the response actions we would take. Specific examples of classified responses that were used in CSM refinement are shown in Appendix A.

General Comments referred to the overly general or abstract nature of the *Principle* or requested the inclusion of more specific Guidelines.

Context Many pointed out that implementation of the *Principle’s Guidelines* would vary according to, for example, programming language, community expectations or personal preference.

Students Respondents had a concern that students would not be able to understand the *Principle* and would need more concrete examples.

Exceptions Some responses discussed situations where an aspect of the *Principle* should not be applied.

Extent Respondents had a concern about ‘taking things too far’ when applying a *Guideline*.

Preference Respondents voiced a preference in how to implement a *Guideline*.

Wording The comment suggests a change in wording to some part of the *Principle*.

Scope Respondents disagree that some aspect of the *Principle* is part of code style.

Meaning Responses indicate that the meaning of one of the *Principle’s* elements is wrong or unclear.

We show the distribution of these themes by *Principle* in Table 10.

Table 10. Distribution of Themes in Open Question Comments by *Principle*

<i>Style element</i>	<i>General</i>	<i>Context</i>	<i>Students</i>	<i>Exceptions</i>	<i>Extent</i>	<i>Preference</i>	<i>Wording</i>	<i>Scope</i>	<i>Meaning</i>
Explanatory Language	1	3	5	1	12	1	8	0	3
Clear Layout	2	3	1	0	0	1	1	0	0
Simple Constructs	1	4	3	2	10	0	1	1	7
Consistent Design	0	1	4	2	0	2	2	3	7
Non-redundant Content	0	1	1	3	2	1	1	1	5
Appropriate Implementation	4	1	3	2	3	0	3	1	14
Avoid Repetition	0	1	1	4	6	1	0	0	7
Modular Structure	1	1	3	2	4	2	4	2	0
Total	9	15	21	16	37	8	20	8	43

6.2 Response Actions

The actions we took in response to the comments are described below:

General As the abstraction level was deliberately selected, we chose not to refine the CSM as a result of the comments. The abstract *Principles* provide the underlying rationale for the many, specific *implementation* guides that have been created by organisations, software projects and individuals according to circumstance and viewpoint. We acknowledge that our intentions may not have been sufficiently clear and believe that some additional support should be provided. We will extend the examples to include example implementations for various contexts.

Context As variations in implementation according to context is in line with our intentions (see *General*), we did not refine the CSM as a result of these comments. However, the frequency of such comments indicates that educators might be uncertain about how to effectively use the CSM. We hope that illustrating with extended examples will provide the relevant support.

Students As the CSM is not intended for direct use by students, we did not refine it as a result of these comments. Our intention is that educators use it as a tool for creating teaching content that suits their needs and goals. For example, a *Principle* might be illustrated by various examples from the available materials and discussed in terms of the underlying rationale. In future, we will create materials to support students directly.

Exceptions We considered these comments on a case-by-case basis. Some resulted in CSM refinement (see Appendix A).

Extent We have recognised that there is a tradeoff in the extent to which a *Principle* is implemented, for example, too many functions might reduce transparency. This is not in conflict with the *Principles*. We did not refine the *Principles* as a result of these comments but we discuss in Section 8.2.7.

Preference Our expectation is that educators will choose concrete implementations of guidelines to support their teaching. Preferences are not in conflict with the intended use of the CSM. We did not refine the CSM as a result of these comments.

Wording We have considered these comments on a case-by-case basis. Many resulted in CSM refinement (see Appendix A).

Scope Our definition of style is intended as a common baseline to support further research but we acknowledge that others may view style differently. We have created the CSM with the expectation that others may adapt it to suit their notion of code style or extend it to

encompass aspects of code quality (see Section 8.2.3). We did not refine the *CSM* as a result of these comments.

Meaning We have considered these comments on a case-by-case basis. Many resulted in *CSM* refinement (see Appendix A).

Significant changes to the *Principle* names were ‘Consistent Design’ was renamed to ‘Be Consistent’, ‘Non-redundant Content’ to ‘No Unused Content’, ‘Appropriate Implementation’ to ‘Congruent Implementation’ and ‘Avoid Repetition’ to ‘Avoid Duplication’. We detail the refinement actions taken in Appendix A.

7 Refined CSM

We show the revised framework below. *Principles* are intended to manifest a *Rationale*. Decisions on how to *implement* the abstract *Guidelines* are not part of the *CSM* and will depend upon context, for example, student experience and programming language. For students in an introductory programming course, some instructors might choose to focus more on learning basic rules and be less strict on style when grading, for example, the use of spacing and choice of data structures. More experienced students might be expected to conform to style guide rules, achieve zero linter violations and choose appropriate structures and algorithms. Illustrative examples of how implementations of the abstract *Guidelines* vary according to programming language are provided below for each *Principle*.

7.1 Definition

Code style is described as ‘*aspects of maintainability, related to the ease of understanding and changing code, that can be determined by reading the source code*’.

7.2 Explanatory Language

Principle The rationale, intent and meaning of code is explicit.

Rationale Being explicit in describing the purpose of the code elements helps us understand the author’s intention, thus improving understandability.

Guidelines

Comments and headers should describe rationale, intent and meaning and be clear and unambiguous to the intended reader.

Names of elements (variables, functions, etc.) should be descriptive and contextually appropriate.

Embedded literals that represent a specific concept should be extracted as named constants that explain the concept.

Implementation Example. Java programmers are advised to use lowerCamelCase for Method and variable names [29]; Python programmers use lowercase with words separated by underscores [30, 54]. The Google Python guide advises that a docstring should give ‘enough information to write a call to the function without reading the function’s code’ [30]. Most guides note that comments must describe rationale and intent.

7.3 Clear Layout

Principle Different elements are easy to distinguish and the relationships between them are apparent.

Rationale Clear layout improves our shared understanding by making the individual elements easy to identify and signalling the elements the author considers to be related.

Guidelines

Formatting (horizontal and vertical spacing, indentation, brackets and line length) should make distinct elements clearly visible.

Placement of elements within a file should highlight the intended structure.

Implementation Example. Python programmers implement 4 spaces per indentation level and line length is limited to 80 characters [54]. Java programmers use 2 spaces per indentation level, a line length of 100 characters [29]. Both languages expect binary operators to be surrounded by a single space on each side.

7.4 Simple Constructs

Principle Coding constructs are selected to minimise complexity for the intended reader.

Rationale Code that is perceived by the reader as simple is easier to understand.

Guidelines

Nesting levels should be minimised.

Flow of control should be easy to follow with few breaks and exceptions.

The size and complexity of expressions should be appropriate for the intended reader.

Implementation Example. Languages often have an implicit ‘preferred’ way of implementing code constructs that can be easily understood by programmers with experience in the language but difficult for other-language programmers to understand. The Google Python guide advises against the general use of lambda functions as these limit expressiveness and are more difficult to understand [30].

7.5 Be Consistent

Principle Elements that are similar in nature are presented and used in a similar way.

Rationale Consistency leverages familiarity to reduce the mental effort required to understand the code.

Guidelines

Documentation and notation should be used consistently throughout.

When making decisions about code, be consistent in implementation choices.

Code should adhere to the programming language idiom.

Implementation Example. Python programmers are advised to be consistent in the choice of string quote character [30, 54], whether to break before or after a binary operator [54], returning expressions for all returns in a function [54], and in the length of comments [30]. The Google Java guide does not discuss consistency [29].

7.6 No Unused Content

Principle All elements that are introduced are meaningfully used.

Rationale Unused code and documentation require unnecessary mental effort.

Guidelines

Unused code and documentation should be removed to reduce mental effort.

Unreachable code should be removed.

Statements that do not affect the logical flow of the program (for example, multiple breaks) should be removed.

Implementation Example. Most language guides do not explicitly discuss removing unused content.

7.7 Congruent Implementation

Principle Implementation choices are consistent with the problem to be solved.

Rationale An implementation that reflects the problem is easier to change.

Guidelines

Implementation choices, such as the structure used, are consistent with the nature of the problem to be solved.

Identifiers and variables should not be reused within the same scope for different purposes.

Implementation Example. PEP-8 advises designing exception hierarchies based on what needs to be caught (i.e. based on the functional requirements) rather than on the location where the exception is defined [54].

7.8 Avoid Duplication

Principle Code duplication is avoided.

Rationale Duplicated code (functions, literals) is difficult to change because there is a risk that not all items are changed.

Guidelines

Duplicated code should be separated out into separate units, e.g., functions and classes.

Embedded literals that represent a specific concept and are used multiple times should be abstracted as constants.

Implementation Example. The Google Python guide advises against comments describing what the code does, thus providing the same information twice [30]. The Google C++ guide advises that, when documenting function overrides, only the specifics of the override should be documented to avoid providing the same information multiple times [28].

7.9 Modular Structure

Principle Related code is grouped together and dependencies between groups are minimised.

Rationale Placing related elements together makes code easier to understand. Reducing inter-connectedness means that isolated pieces can be more easily understood and can be modified independently.

Guidelines

Modules should be created with clearly defined responsibilities and minimal dependence on other modules.

Functions should aim to implement a single task and have a minimum number of parameters and side effects.

Code should be organised such that the most related elements appear together.

The scope of constants and variables should be minimised.

Implementation Example. PEP-8 advises that functions should aim to be free of side effects with the exception of caching [54]. Google Python states that getters and setters should be used when a change of state of an internal attribute is involved [30]. Mutable global variables should be avoided. PEP-8 advises that care should be taken when deciding the visibility of class attributes [54].

8 Findings

8.1 RQ

8.1.1 RQ. How Might We Reconcile the Vagueness and Inconsistencies Around Discussions on Code Style? We created, evaluated and refined a conceptual model that encapsulates the code-level

attributes that represent ‘good’ code style. We created objectives for each phase that would provide us with confidence in the validity of the model.

8.1.2 RO 1. Clearly Articulate the Process of Model-Building. We have presented the main steps taken when building the CSM in Sections 3.2 and 4 and have described these more fully in prior work [41].

8.1.3 RO 2. Evaluate the Level of Community Support for the Principles. Our success in mapping style guidelines and ‘Clean Code’ heuristics to the *Principles* (see Tables 4 and 5) gives us confidence that the CSM captures the key code-level attributes of code style. The exposure of the ‘non-code-based’ nature of some advice given as ‘Clean Code’ heuristics provides us with confidence that the CSM might be useful with respect to addressing the issue of vague terminology as discussed in Section 2.4.3.

The high representation of positive Likert responses shown in Figures 4 and 5 indicates an overall endorsement of the *Style Definition* and *Principles*. This is further illustrated in Table 7. Five of the eight *Principles* achieved $\geq 84\%$ agreement for all of *understanding the description of the Principle*, *understanding and changing code that complies with the Principle* and *the support provided by the Principle for teaching*. For the open-question responses, 49 voiced disagreement with some aspect of the model (see Table 8). Analysis showed that many of these related to scope or the contextual nature of style. We believe that those who take the time to make comments tend to focus on aspects they feel strongly about and the implications are that people hold strong viewpoints about style and relate style to a specific context. Very few comments were associated with disagreement about a core aspect of the model.

These findings indicate a high level of support and endorsement for the approach we have taken and for the *Principles* described by the CSM. We discuss the less highly-rated *Principles* (*Simple Constructs*, *Be Consistent* and *Congruent Implementation*) in Section 9.

8.1.4 RO 3.1. Justify the Changes Made to the CSM as a Result of Recommendations Made by the CSE Community. There were 177 suggestions for change, grouped into 9 categories (see Table 10). Suggestions in 6 of the categories resulted in no change to the CSM. Categories *General*, *Context* and *Students* indicated a misunderstanding of our intent. These highlighted a lack of clarity in our intentions and exposed the need for a more comprehensive set of examples. Categories *Preference* and *Scope* contained feedback that stated individual viewpoints on style content and implementation. These are not inconsistent with our intentions. Category *Extent* is not in conflict with the CSM but did cause us to think more deeply about the need for tradeoffs when deciding ‘how much is enough’ (see Section 8.2.7). Comments relating to *Exceptions to a Principle* and the *Wording* and *Meaning* of a *Principle* resulted in the changes described in Appendix A.

8.1.5 RO 3.2. Identify the Issues that Arise When Attempting to Develop a Community-Wide CSM. Beliefs about what constitutes style and how style should be implemented are strongly held by many as ‘objective truths’ when in fact these are subjective opinions, formed within a specific context and unsupported by evidence. Until the community recognises this, it will be difficult to get an overall agreement on a CSM.

A second barrier lies in the abstract nature of the CSM *Principles*. This made it difficult to grasp the notion that a *Principle* might support implementation in many contexts, for example, different programming languages and programmer experience.

8.2 Discussion

Our goal is not to change existing style advice or propose new advice—our goal is to explain and reconcile existing advice. For example, the *Principle* Clear Layout covers advice for how to make

individual elements in code easy to identify. In this light, the example of use of whitespace given in the introduction becomes less confusing—in some places, whitespace is not needed to help identify individual elements and in other cases, the amount of whitespace depends on the particular context (operators and precedence). In this case, such rationale could be guessed from the examples given, but this is not universally the case. Furthermore, the *CSM* provides a more rigorous basis for such discussions.

The results from our survey were generally very positive and this has convinced us that the direction we are taking is appropriate. Changes to the original *Principles*, while relatively small, have led to significant improvements, giving us the confidence to present what we now regard as a *CSM*. The feedback we received led us as researchers to develop a deeper understanding that has clarified some important aspects relating to our vision for model use. We discuss these below.

8.2.1 Abstraction. Many participants' comments referred to the overly general nature of the *Principles*, pointed out the dependence on context or requested the inclusion of specific implementations. It is the huge range of possible contexts and implementations that has caused it to be difficult for educators to decide what to teach and for students to understand what is 'good style'. Our choice of abstraction level is deliberate and provides a mechanism for making sense of local variations.

However, some comments suggested that the abstract nature of the *Principles* renders them of limited practical value as-is. Educators are busy and creating a teaching strategy based on the *CSM* requires time and effort. In future work, we will create sets of examples and draft teaching plans based on the *CSM*.

8.2.2 Context Dependence. Style rules and conventions are created according to context, often with the aim of promoting consistency among developers. The most obvious context is programming language. For example, Google has different style guides for each of its development languages [28–30]. The Python PEP-8 Style Guide states that 'Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project' [54]. Several respondents observed that reader characteristics, for example, experience, also affect style choices. For example, highly commented code may be appropriate for novices but viewed as overly fussy by experienced programmers. Crossley et al. found that text familiarity was important when investigating text comprehension in adults, and suggested that scaffolding should involve 'matching text difficulty to the abilities of the reader' [18]. Schulte et al. found that prior knowledge influences a reader's ability to comprehend programs [59]. Wiese et al. found that novices did not always find expert-style code easier to comprehend and concluded that students 'do not acquire the skills of recognition, usage, comprehension ... at the same time' [73]. The *Guidelines* included in the *CSM* are consistent with these findings as they are to be adapted as needed.

8.2.3 Included and Excluded. Of the 84 respondents, the Likert responses for 5 (6%) indicated disagreement with our *Definition* of style (see Section 5.1). Of the 37 responses to the open question on style definition, only 3 addressed scope. One suggested that modularity and structure are related to *quality* whereas style was 'more about choice, like whether braces go on the same line or next ...'. Two believed that runtime efficiency should be included ('efficiency ... is something we emphasise as part of style in our introductory classes').

We recognise that what should be considered as style is not fixed. However, the survey data implies that our scoping is generally acceptable. Our intention is simply to come up with an agreed, workable definition that can be used by the research community as a starting point for discussion and teaching. Those who disagree are free to extend the *Definition* and include *Principles* that are relevant for the additional quality characteristics.

8.2.4 Intended CSM Usage. A small number of respondents noted that the abstract nature of the guidelines would leave them inaccessible to students and more concrete guidance is needed. The CSM is not intended for direct use by students. Providing concrete guidance in the model would leave students in the same situation as the present one, i.e., trying to make sense of many different, conflicting rules. The intention is that educators use the CSM as a basis to create teaching plans according to their need. For example, ‘Explanatory Language’ might be presented in terms of *why* this matters (relating back to maintainability) and conflicting rules reconciled. Educators are free to choose rules based on the desire to illustrate a specific point or personal preference.

8.2.5 Holism. As each *Principle* is motivated by a different *Rationale*, style rules can be analysed in a holistic way, according to underlying purpose. This raises the need to be clear about purpose when analysing a style rule (see Section 8.2.8). The holistic nature of the CSM represents an opportunity to program designers in that courses might be designed such that students are exposed to *Principles* within an over-arching plan. For each course, *Principles* might be selected according to course learning outcomes and courses organised to cover all aspects over the program. Specific examples would be chosen according to educator preferences or course constraints.

The CSM focuses on understanding and changing code. Our claim is that the *Principles* represent a sufficient and complete set with respect to the target product quality characteristics *Understandability* and *Modifiability*. However, they are clearly not a complete set when considered with respect to other product quality characteristics, for example, *Run-time Efficiency*. Those who prefer to include other characteristics in code style must create an auxiliary set of *Principles* to address these characteristics. We believe our model provides a suitable basis for such extensions.

8.2.6 Tradeoffs. CSM users may be required to choose between the relative importance of two *Principles* when these conflict. For example, *Simple Constructs* requires that code complexity be reduced and *Be Consistent* requires that code should adhere to the programming language idiom. The need for a tradeoff between perceived simplicity and familiarity may occur for programs written in ‘C’ when the concise nature of the code may appear simple to another ‘C’ programmer (the expected reader) but complex to a Java programmer tasked with porting the code to Java. Tradeoffs also may be required between style and other quality characteristics. For example, a need for runtime efficiency may involve non-compliance with the *Modular Structure Principle*. Tradeoffs should be dealt with on a case-by-case basis. Decisions would ideally be part of the specification process, during discussions on expected usage and the relative importance of different product quality characteristics. As each *Principle* is grounded in a *Rationale*, this represents a rich area for future exploration.

8.2.7 Sliding Scale. Feedback caused us to consider the question of ‘how much is enough’. For *Explanatory Language*, how much commenting is ‘good’ and how long should variable names be? For *Simple Constructs*, is it acceptable to have a return in the middle of a loop, or should there be only a single return at the end? For *Modular Structure*, how small should functions be? Will increasing modularity lead to reduced transparency when long reference chains must be followed to establish how a function actually behaves?

The literature includes many, heated discussions about these kinds of questions. It seems likely that ‘how much is enough’ depends on aspects of the reader (for example, students versus professional developers) and application type (for example, architectural considerations relating to change). We believe the exploration of this topic represents an important avenue for research and that the CSM will provide a theoretical basis upon which to base future studies.

8.2.8 Basis in Purpose. Some comments indicated a confusion about which *Principle* a code feature belongs to. *Principles* are based on different *Rationales* and so form a mutually exclusive set.

However, some code features encompass multiple ‘ideas’, each of which associates with a different *Principle*. An example is that of embedded literals. The goal of making it clear what a literal *means* is addressed by *Explanatory Language*, where the advice is that a ‘magic number’ should be extracted to a descriptively named constant. The goal of making code *easier to change* is addressed by *Avoid Repetition*, where the advice is to replace repeated code by a single instance. A consequence is that a multiple occurrence of an embedded literal must be analysed in terms of both *Explanatory Language* and *Avoid Repetition*.

We believe that making these distinctions based on purpose is essential for progressing with formal studies on code style. We cannot build strong systems upon shaky foundations where the categories used as a basis are not clearly defined, with possibilities for overlap.

8.2.9 Style versus Quality. Our analysis of the related literature (see Section 2) exposed a lack of clarity around the terms ‘style’ and ‘quality’. We believe that clarity around these terms is essential for progress with research to support teaching. We have investigated this further in a parallel study [40] and plan to continue with these investigations in future works.

8.2.10 Model to Support Research. According to Gilmore, ‘A vital component in any hypothesis testing research is the theoretical framework, since this both enables the researcher to make decisions about what tasks to provide and what data to measure and it provides the context within which the results can be explained. Hypothesis-testing research is not possible without a theoretical framework ...’ [27]. Basili et al. believed that, when carrying out controlled experiments, ‘... it is hard to know how to abstract important knowledge without a framework for relating the studies’ [6].

We believe that the *CSM* has the potential to be regarded as one of a set of theoretical models to support code complexity studies. The claim for the *Principles* is that they describe attributes that represent the inherent readability of code. We see parallels with the education sector, where readability is described by four complexity levels, i.e., lexical, syntactic, semantic and discourse complexity. In future work, we will explore the appropriateness of the *CSM* as one element of a research model by mapping the *Principles* to these complexity levels.

9 Threats and Limitations

The software product models from which we sourced the information do not represent an exhaustive search of the literature and it is possible that more recent models would have provided other insights. However, these are the most commonly cited models and we do not believe the basic product quality (sub) characteristics have changed over time. In addition, we have included the ISO/IEC Standard 25010 [33], considered to be the authoritative model for software product quality.

The differences in perspective exhibited in the product quality models from which the software *Principles* were derived meant that some pragmatic decisions had to be made about what to include. For example, the Dromey model includes some construct-specific notions, such as initialising variables prior to use and coupling considerations. We consider these are more low level considerations and we did not include them as *Principles*.

Although the questionnaire was distributed to a membership list that represents 66 countries, we did not collect demographics on the nationality of respondents and so we do not know the spread of nationalities represented by the respondents. This means that we may have missed some interesting perspectives. Distribution to an academic mailing list also means that we do not have insights into how professional developers view the *CSM*. As evidence suggests that industrial and academic programmers focus on different aspects of style [10], our findings may not be transferable to professional developers. We do not believe that this is a significant limitation because the goal of

this research was to support academic instructors. In future work, we plan to evaluate the CSM in an industry context.

Some aspects of the CSM require further investigation. In Section 8, we discussed the lower levels of acceptance for parts of the *Principles Simple Constructs, Be Consistent* and *Congruent Implementation*. These must be revisited. There is uncertainty around the applicability of the *Modular Design Principle* when considering larger programs. We believe the CSM is paradigm-agnostic because of the basis in *rationale* but have yet to confirm that it applies to non-OO paradigms. We have positioned the CSM as addressing understanding and changing code. The *Rationale* for each *Principle* is based on how the *Principle* affects these. Rationales have been based loosely on underlying models for understanding and change. We plan to formalise these relationships.

The level of abstraction for the *Principles*, described in Section 4.1 and illustrated in Figure 1, represents a pragmatic choice based on terms used in product quality models to describe the *software* as opposed to the *product*. McCall et al. discussed the code criteria ‘modularity’ and ‘complexity’ [43] and Dromey discussed the ‘quality carrying properties’ of code [22]. This choice of level has not yet been tested in the real world and we expect that the *Principles* will evolve with exposure. One outcome might be that some *Principles*, for example, those with a lower acceptance rate (see Figure 5), are overly complex, and this possibly indicates that the abstraction level needs to be modified. This may involve splitting some *Principles*. We also suspect that some structure around abstraction levels will prove to be necessary, for example, grouping *Principles* that contribute to *code* ‘readability’. We will investigate these concerns in future studies.

The quality focus for this study was maintainability, in particular, on understandability and changeability. However, as noted in Section 2.4, many practitioners and educators associate other quality characteristics, such as efficiency and correctness, with style. We have recognised that this potentially limits the usefulness of the CSM and have suggested, in Section 8.2.3, that *Principles* might be extended to address other qualities. In future work, we plan to extend the *Principles* and re-position the model as a *Code Quality Model*.

We have created, evaluated and refined the CSM based on software product quality models and feedback from educators. Our claim is that the revised CSM will be useful to the teaching community. However, the subjective nature of the study means that we will only know whether this is a valid claim if the CSM is accepted and adopted by the CSE community. The results of our survey are very encouraging. Our next step is to evaluate the CSM within the classroom.

10 Summary

Educators would like to prepare students for working in the software industry by teaching students how to write code with ‘good style’. Many style guides and standards exist but these vary in content and offer conflicting advice. The inconsistencies in the available guides leave many educators struggling to explain to students what style is, why it matters and why one style rule is better than another.

In this article, we present the CSM, CSM, that addresses the inconsistencies in style guides by focusing on the *rationale* behind style rules. The objective is to support educators in helping students understand *why* a rule exists and its contextual nature. The CSM contains a style *Definition* and eight abstract *Principles*. We applied an exploratory approach to create the CSM and distributed a survey questionnaire to a large, international population of CS educators to evaluate and refine it.

Findings indicated a strong level of support for the CSM. Quantitative data showed a high level of agreement with the *Principles* and qualitative data contained very few comments that voiced dissatisfaction with a basic aspect of the model. We found that barriers to developing a community-wide CSM were the strongly-held beliefs about style held by some and the lack of specific guidance as a result of the abstract nature of the CSM.

In future work, we will create teaching plans for specific contexts based on the *Principles* and will investigate and further refine the *Principles* that were less highly rated by respondents. The study highlighted a lack of clarity in the use of the terms ‘quality’ and ‘style’. We believe the vague use of these terms does not support rigorous research and will address this in future work.

The CSM has been endorsed by the community. We believe it can be adapted and used by educators in the classroom to facilitate discussions on code style.

References

- [1] Joe Michael Allen, Frank Vahid, Alex Edgcomb, Kelly Downey, and Kris Miller. 2019. An analysis of using many small programs in CS1. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, 585–591. DOI : <https://doi.org/10.1145/3287324.3287466>
- [2] Eric Allman. 2012. Managing technical debt. *Communications of the ACM* 5, 5 (2012), 50–55. DOI : <https://doi.org/10.1145/2160718.2160733>
- [3] David Armstrong, Ann Gosling, John Weinman, and Theresa Marteau. 1997. The place of inter-rater reliability in qualitative research: An empirical study. *Sociology* 31, 3 (1997), 597–606.
- [4] Association for Computing Machinery. 2024. SIGCSE Profile. Retrieved June 7, 2024 from <https://www.sigcse.org/about/profile.html>
- [5] S. K. Baker, F. Fien, N. J. Nelson, Y. Petscher, S. Sayko, and J. Turtura. 2017. Learning to Read: The Simple View of Reading. Retrieved September 15, 2023 from <https://improvingliteracy.org/brief/learning-read-simple-view-reading>
- [6] Victor R. Basili, Forrest Shull, and Filippo Lanubile. 1999. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering* 25, 4 (1999), 456–473.
- [7] 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 (ITiCSE '23)*. ACM, New York, NY, 152–158. DOI : <https://doi.org/10.1145/3587102.3588800>
- [8] B. W. Boehm, J. R. Brown, and M. Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*. IEEE Computer Society Press, 592–605.
- [9] Jürgen Börstler, Kwabena E. Bennin, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, Bonnie MacKellar, Rodrigo Duran, Harald Störrle, Daniel Toll, and Jelle van Assema. 2023. Developers talking about code quality. *Empirical Software Engineering* 28, 128 (2023). DOI : <https://doi.org/10.1007/s10664-023-10381-0>
- [10] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2017. “I know it when I see it” Perceptions of code quality: ITiCSE '17 working group report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports (ITiCSE-WGR '17)*. ACM, New York, NY, 70–85. DOI : <https://doi.org/10.1145/3174781.3174785>
- [11] Shreya Bose. 2021. Coding Standards and Best Practices to Follow. Retrieved May 12, 2023 from <https://www.browserstack.com/guide/coding-standards-best-practices>
- [12] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. DOI : <https://doi.org/10.1191/1478088706qp063oa>
- [13] Cambridge Assessment International Education (CAIE). 2019. Syllabus: Cambridge International AS & A Level Computer Science. Retrieved October 7, 2019 from <https://www.cambridgeinternational.org/Images/202629-2017-2019-syllabus.pdf>
- [14] Cambridge University Press. 2024. Syntax. Retrieved October 9, 2024 from <https://dictionary.cambridge.org/dictionary/english/syntax>
- [15] CISQ. 2023. Coding Rules for Maintainable Software. Consortium for Software & Information Quality (CISQ). Retrieved May 12, 2023 from <https://www.it-cisq.org/standards/code-quality-standards-maintainability/>
- [16] Paul Clarke, Antoni-Lluís Mesquida, Damjan Ekert, J. J. Ekstrom, Tatjana Gornostaja, Milos Jovanovic, Jørn Johansen, Antonia Mas, Richard Messnarz, Blanka Nájera Villar, Alexander O'Connor, Rory V. O'Connor, Michael Reiner, Gabriele Sauberer, Klaus-Dirk Schmitz, and Murat Yilmaz. 2016. An investigation of software development process terminology. In *Proceedings of the Communications in Computer and Information Science (CCIS '16)*, Vol. 609. Springer International Publishing, Switzerland, 351–361.
- [17] John W. Creswell. 2014. *The Selection of a Research Approach*. Sage Publications, Inc., 31–55.
- [18] Scott A. Crossley, Stephen Skalicky, Mihai Dascalu, Danielle S. McNamara, and Kristopher Kyle. 2017. Predicting text comprehension, processing, and familiarity in adult readers: new approaches to readability formulas. *Discourse Processes* 54, 5–6 (2017), 340–359. DOI : <https://doi.org/10.1080/0163853X.2017.1296264>
- [19] Tom Dalling. 2009. Coding Tip: Have A Single Exit Point. Retrieved June 22, 2023 from <https://www.tomdalling.com/blog/coding-tips/coding-tip-have-a-single-exit-point/>

- [20] J. W. de Bakker. 1969. *Semantics of Programming Languages*. Springer US, Boston, MA, 173–227. DOI : https://doi.org/10.1007/978-1-4899-5841-9_3
- [21] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. 2018. Understanding semantic style by analysing student code. In *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*. ACM, New York, NY, 73–82. DOI : <https://doi.org/10.1145/3160489.3160500>
- [22] R. G. Dromey. 1995. A model for software product quality. *IEEE Transactions on Software Engineering* 21, 2 (1995), 146–162.
- [23] R. G. Dromey. 1996. Cornering the Chimera. *IEEE Software* 13, 1 (1996), 33–43.
- [24] S. Easterbrook, J. Singer, M.A. Storey, and D. Damian. 2008. Selecting empirical methods for software engineering research. In *Guide to Advanced Empirical Software Engineering*. F. Shull and J. Singer and D. I. K. Sjøberg (Ed.), Springer International Publishing, London, UK, 285–311.
- [25] Martin Fowler. 2009. Technical Debt Quadrant. Retrieved October 3, 2024 from <https://martinfowler.com/bliki/TechnicalDebtQuadrant.html>
- [26] Nicola Gale, Gemma Heath, Elaine Cameron, Sabina Rashid, and Sabi Redwood. 2013. Using the framework method for the analysis of qualitative data in multi-disciplinary health research. *BMC Medical Research Methodology* 13, 117 (2013). DOI : <https://doi.org/10.1186/1471-2288-13-117>
- [27] David J. Gilmore. 1990. Methodological issues in the study of programming. In *Psychology of Programming*. J.-M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore (Eds.), Academic Press Ltd., London, U.K., 83–98.
- [28] Google. 2023. Google C++ Style Guide. Retrieved May 12, 2023 from <https://google.github.io/styleguide/cppguide.html>
- [29] Google. 2023. Google Java Style Guide. Retrieved May 12, 2023 from <https://google.github.io/styleguide/javaguide.html>
- [30] Google. 2023. Google Python Style Guide. Retrieved May 12, 2023 from <https://google.github.io/styleguide/pyguide.html>
- [31] Google. 2022. Google Style Guides. Retrieved January 11, 2022 from <https://google.github.io/styleguide/>
- [32] International Organization for Standardization. 2013. *ISO/IEC 25010: Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models*. ISO/IEC.
- [33] International Organization for Standardization. 2016. *ISO/IEC 25023: Systems and software Quality Requirements and Evaluation (SQuaRE)—Measurement of system and software product quality*. ISO/IEC.
- [34] B. W. Kernighan and P. J. Plauger. 1974. Programming style. In *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '74)*. ACM, New York, NY, 90–96. DOI : <https://doi.org/10.1145/800183.810448>
- [35] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, 119–125. DOI : <https://doi.org/10.1145/3304221.3319780>
- [36] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2023. A systematic mapping study of code quality in education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023)*. ACM, New York, NY, 5–11. DOI : <https://doi.org/10.1145/3587102.3588777>
- [37] Diana Kirk. 2021. Software development context: critiquing often used terms. In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE) 2021*. Online Event, 340–347. DOI : <https://doi.org/10.5220/0010469903400347>
- [38] Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. 2020. On Assuring Learning about Code Quality. In *Proceedings of the 22nd Australasian Computing Education Conference (ACE'20)*. ACM, 86–94. DOI : <https://doi.org/10.1145/3373165.3373175>
- [39] Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. 2022. Teaching code quality in high school programming courses—Understanding teachers' needs. In *Proceedings of the Australasian Computing Education Conference (ACE '22)*. ACM, New York, NY, 36–45. DOI : <https://doi.org/10.1145/3511861.3511866>
- [40] Diana Kirk, Andrew Luxton-Reilly, and Ewan Tempero. 2024. Code style != Code quality. In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1 (SIGCSE Virtual '24)*. ACM, New York, NY, 267–270. DOI : <https://doi.org/10.1145/3649165.3703621>
- [41] 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 (ACE '24)*. ACM, New York, NY, 134–143. DOI : <https://doi.org/10.1145/3636243.3636258>
- [42] Robert C. Martin. 2009. *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice-Hall, Inc.
- [43] Jim A. McCall, Paul K. Richards, and Gene F. Walters. 1977. *Factors in Software Quality*. Technical Report RADC-TR-77-369, Volume 1 (of three). General Electric Company.
- [44] José P. Miguel, David Mauricio, and Glen Rodríguez. 2014. A review of software quality models for the evaluation of software products. *International Journal of Software Engineering & Applications* 5, 6 (2014), 31–54.
- [45] John C. Mitchell. 1996. *Foundations for Programming Languages*. MIT Press, Cambridge, Mass.
- [46] Monash University. 2021. Programming Styles. Retrieved May 12, 2023 from <https://www.monash.edu/it/current-students/resources-and-support/style-guide/programming-styles>

- [47] Sara Nurollahian, Anna N. Rafferty, Noelle Brown, and Eliane Wiese. 2024. Growth in knowledge of programming patterns: A comparison study of CS1 vs. CS2 students. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE '24)*. ACM, New York, NY, 979–985. DOI: <https://doi.org/10.1145/3626252.3630865>
- [48] New Zealand Ministry of Education. 2017. The New Zealand Curriculum. Retrieved December 9, 2021 from <http://nzcurriculum.tki.org.nz/The-New-Zealand-Curriculum>
- [49] John Ousterhout. 2018. *A Philosophy of Software Design*. Yaknyam Press, Palo Alto, CA.
- [50] Patrick E. Parrish. 2009. Aesthetic principles for instructional design. *Educational Technology Research and Development* 57 (2009), 511–528. DOI: <https://doi.org/10.1007/s11423-007-9060-7>
- [51] Marc Pierson. 2014. Single Function Exit Point. Retrieved June 22, 2023 from <https://wiki.c2.com/?SingleFunctionExitPoint>
- [52] Catherine Pope, Sue Ziebland, and Nicholas Mays. 2000. Qualitative research in health care: Analysing qualitative data. *British Medical Journal* 320 (2000), 114–116.
- [53] Pull Request. 2021. How to Create a Programming Style Guide. Retrieved November 23, 2021 from <https://www.pullrequest.com/blog/create-a-programming-style-guide/>
- [54] Python Software Foundation. 2023. PEP-8 Style Guide for Python Code. Retrieved November 17, 2023 from <https://peps.python.org/pep-0008/>
- [55] Anna Řečtáčková, 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 (ITiCSE '24)*. ACM, New York, NY, 59–65. DOI: <https://doi.org/10.1145/3649217.3653638>
- [56] T. Richards and L. Richards. 1995. Using hierarchical categories in qualitative data analysis. *Computer-Aided Qualitative Data Analysis: Theory, Methods, and Practice* (1995), 80–95.
- [57] Pentti Routio. 2007. Models in the Research Process. Retrieved from <http://www2.uiah.fi/projects/metodi/177.htm>
- [58] Ognjen Savkovic. 2015. Good Coding Style. Retrieved May 12, 2023 from <https://www.inf.unibz.it/~nutt/Teaching/DSA1819/DSAAssignments/good-coding-style.html>
- [59] 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 (ITiCSE-WGR '10)*. ACM, New York, NY, 65–86. DOI: <https://doi.org/10.1145/1971681.1971687>
- [60] Shaindel Schwartz. 2020. *Style Guides and Rules*. O'Reilly Media, Inc., CA, 141–153.
- [61] Darja Šmite, Claes Wohlin, Zane Galviņa, and Rafael Prikladnicki. 2014. An empirically based terminology and taxonomy for global software engineering. *Empirical Software Engineering* 19 (2014), 105–153. DOI: <https://doi.org/10.1007/s10664-012-9217-9>
- [62] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 595–609. DOI: <https://doi.org/10.1109/TSE.1984.5010283>
- [63] Martijn Stegeman. 2016. Code Quality. Retrieved January 11, 2022 from <https://www.stgm.nl/quality/>
- [64] 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*. ACM, New York, NY, 99–108. DOI: <https://doi.org/10.1145/2674683.2674702>
- [65] 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 Calling '16)*. ACM, New York, NY, 160–164. DOI: <https://doi.org/10.1145/2999541.2999555>
- [66] Klaas-Jan Stol and Brian Fitzgerald. 2015. Theory-oriented software engineering. *Science of Computer Programming* 101 (2015), 79–98.
- [67] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of software engineering research. *ACM Transactions on Software Engineering and Methodology* 27, 3 (2018), 11.1–11.51.
- [68] The Apache Software Foundation. 2015. Coding Standards. Retrieved May 19, 2023 from <https://ace.apache.org/docs/coding-standards.html>
- [69] Maria Tigina, Anastasiia Birillo, Yaroslav Golubev, Hieke Keuning, Nikolay Vyahhi, and Timofey Bryksin. 2023. Analyzing the quality of submissions in online programming courses. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '23)*, 271–282. DOI: <https://doi.org/10.1109/ICSE-SEET58685.2023.00031>
- [70] Adam Tornhill. 2023. Business Costs of Technical Debt. Whitepaper. Code Scene. Retrieved October 9, 2023 from <https://codescene.com/hubfs/calculate-business-costs-of-technical-debt.pdf>
- [71] Venngage Inc. 2023. Accessible Color Palette Generator. Retrieved August 15, 2023 from <https://venngage.com/tools/accessible-color-palette-generator#colorGenerator>
- [72] W3C. 2010. HTML Usage Guide/Syntax-Semantics. Retrieved November 23, 2021 from <https://www.w3.org/html/wg/wiki/Guide/Syntax-Semantics>

- [73] Eliane S. Wiese, Anna N. Rafferty, Daniel M. Kopta, and Jacquelyn M. Anderson. 2019. Replicating novices' struggles with coding style. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. IEEE Press, 13–18. DOI: <https://doi.org/10.1109/ICPC.2019.00015>
- [74] Wikipedia. 2023. Programming Style. Retrieved May 12, 2023 from https://en.wikipedia.org/wiki/Programming_style
- [75] Joseph M. Williams. 1990. *Style: Toward Clarity and Grace*. The University of Chicago Press, Chicago and London.
- [76] Nicholas C. Zakas. 2023. Why Coding Style Matters. Retrieved May 12, 2023 from <https://www.smashingmagazine.com/2012/10/why-coding-style-matters/>

Appendix

A CSM Refinement Detailed Description

Below is a detailed analysis of the open survey comments that resulted in a change to the CSM or provided important insights. For the distribution of comments by theme, see Table 10. For the refined CSM, see Section 7.

A.1 Code Style Definition

The definition for code style is *'aspects of software quality that can be determined by looking at the source code, constrained to understanding and changing code'*.

Maintainability Two participants suggested replacing 'changing' to 'maintaining' and one limited style to understanding only (*I think about style more as an aid to comprehension than to changing code*). We re-examined the common software product quality models. The Boehm et al. model identifies 'understandability', 'modifiability' and 'testability' as sub-characteristics of 'maintainability' [8]. In the McCall model, product quality characteristics are given as factors and there is no notion of sub-factors. However, the model cites lower, code-level criteria for the 'maintainability' factor that include 'consistency' 'self-descriptiveness' and 'modularity' [43] and we recognise that these are generally viewed as supporting understanding and changing. The ISO/IEC Standard for System and Software Quality Models, ISO/IEC 25010, defines 'maintainability' as the 'degree of effectiveness and efficiency with which a product or system can be modified by the intended readers' [32] and includes the sub-characteristic 'modifiability'. A consideration of the above causes us to believe that changing is a *necessary pre-requisite* to, and is *not synonymous* with, maintaining. Our response was to leave this aspect of the definition as-is.

Quality Four responses indicated an issue around the term 'quality'. These related to the meaning of the term (*I think the definition provided depends on that term, and I'm not sure it is consistently understood*) and the association between style and quality (*not clear code style is an aspect of software *quality**). From these responses, we understood that positioning the framework within the software quality landscape had generated some confusion and was not helpful. Product quality has many aspects, [8, 32, 43]. As our intention was to position code style within maintainability, our response was to focus the definition by replacing 'quality' with 'maintainability'.

Scope Five respondents disagreed with our scoping of style, for example, to exclude efficiency (*but would also add run time efficiency*). We acknowledge that there are many viewpoints on what style is (see Section 8.2.3). Some believed our definition implied other quality characteristics (*I can potentially judge a program's efficiency by looking at the code, for example, but I don't typically think of that as a style issue*). We hope that the modified definition will make it more clear that we restrict style to maintainability.

Context Six respondents focused on the importance of a consistent, agreed convention to support teamwork (*I prefer to discuss conventions and provide the context that because most software is developed in team*). Two noted the personal aspect of style (*I guess what I feel is*

missing, is wanting to go towards the ‘personalised’ aspect of code style ... this definition doesn’t quite capture the ‘freedom’ aspect of code style, and sort of implies the code style ‘is just there, inherent in the code’ without any regard to the programmer’s role in it being there). We agree with both points and expect that specific rules will be chosen according to team needs and preferences. We are comfortable that the generic, non-specific definition is appropriate and discuss contextual aspects in Section 8.2.2.

Design Two respondents discussed scaling up to large, complex systems (*It depends on the level of engagement with the code. E.g. a complex program with multiple classes might have its quality defined in terms of the design and code i.e. coupling and cohesion*). We acknowledge that there is a ‘grey area’ when considering style-in-the-large (see Section 9).

Language Four responses commented on the language used. Two concerned the word ‘looking’ (*I don’t think that ‘looking’ is the best verb here. ‘Examining’ or something conveying a range of tasks from reading to reflecting is closer to what I would mean*) and two concerned the phrase ‘understanding and changing code’ (*I don’t know what ‘understanding and changing code’ means—not even sure that’s grammatically correct*). Our responses were a) to replace ‘looking’ by ‘reading’ as this ‘involves a complex integration of skills’ [5], i.e., implies something deeper than simply looking and b) to rework the definition to make it more grammatically correct and hopefully clearer.

A.2 Code Style Principle—Explanatory Language

The Explanatory Language Principle is:

Principle The intent and meaning of code is explicit.

Rationale Being explicit in describing the purpose of the code elements helps us understand the author’s intention, thus improving understandability.

Guidelines

Comments and headers should describe intent and meaning and be grammatically correct and written in correct English.

Names of elements (variables, functions, etc.) should be descriptive and contextually appropriate.

Embedded literals should be extracted as named constants.

Exceptions One respondent observed that *0 and 1 are embedded literals, but, when used naturally, probably shouldn’t be extracted as constants*. We acknowledge that embedded values that are used, for example, to initialise a count should remain as-is. We have changed the wording of the third Guideline to include ‘that represents a specific concept’.

Wording Four responses disagreed that comments be written in grammatically correct English. Reasons given were that grammatically correct does not always equate to clear (‘*Grammatically correct*’ could be interpreted as ‘*fluent*’ rather than ‘*clear*’ or ‘*unambiguous*’), might penalise non-native-speaking students and does not support the rights of indigenous communities. The Guideline was based on the understanding that English is the accepted language for sharing information within the software community. However, we accept the feedback and have modified the first Guideline. We agree with the suggestion to *explain why a choice was made (e.g., to reveal a connection with other code)—not just the intent of the code*. and have extended the Principle to include ‘rationale’. For the suggestion to *add ‘to provide explanatory names’ to make it clear why using constants can make code more clear*., we have extended the third Guideline. One suggestion was to include advice that *‘header comments should match the current implementation’*. This relates to *consistency* within a file and is addressed

by the Consistent Design Principle. We discuss the importance of making these distinctions in Section 8.2.8.

A.3 Code Style Principle—Clear Layout

The Clear Layout Principle is:

Principle Different elements are easy to distinguish and the relationships between them are apparent.

Rationale Clear layout improves our shared understanding by making the individual elements easy to identify and signalling the elements the author considers to be related.

Guidelines

Formatting (horizontal and vertical spacing, indentation, brackets and line length) should make distinct elements clearly visible.

Placement of elements in a file should highlight the intended structure.

Context The comment *should match the conventions of the language and codebase* is addressed by the Consistent Design principle (see Section 8.2.5).

Wording One respondent suggested (*I would use the word 'structure' to describe the rationale behind this principle*). We have not implemented the suggestion as we believe this might cause confusion with the Modular Structure Principle (see Section A.9).

A.4 Code Style Principle—Simple Constructs

The Simple Constructs Principle is:

Principle Coding constructs are implemented in a way that minimises complexity for the intended reader.

Rationale Simple code is easier to understand.

Guidelines

Nesting levels should be minimised.

Flow of control should be easy to follow with few breaks and exceptions.

Expression density in statements should be appropriate for the intended reader.

Extent Ten respondents noted the tradeoffs needed when considering, e.g., nesting levels and break statements. We discuss tradeoffs *within* a Principle in Section 8.2.7.

Exceptions Two responses pointed out that exceptions might help with making intent more clear *Flow of control can interfere with the concept of keeping logic concepts together*. This describes a tension with Clear Layout. We acknowledge that trade-offs between Principles may be necessary and discuss this in Section 8.2.6.

Wording One response suggested that 'implemented' was not the correct term to use for choice of constructs. We have reworded as 'selected to minimise'.

Meaning Three respondents noted that the term 'simple' is subjective (*there's just too much subjectivity involved with what's considered 'simple'*). We agree and have extended the Rationale to emphasise this reader-dependence. We discuss the contextual nature of the CSM in Section 8.2.2. Four respondents were uncertain about the meaning of 'expression density'. We have replaced the term in the third Guideline with 'size and complexity of expressions'.

A.5 Code Style Principle—Consistent Design

The Consistent Design Principle is:

Principle Elements that are similar in nature are presented and used in a similar way.

Rationale Consistency leverages familiarity to reduce the mental effort required to understand the code.

Guidelines

Documentation and notation should be used consistently throughout.

Design techniques should be applied consistently throughout.

Code design and constructs should adhere to the programming language idiom.

Exceptions One respondent noted *I sometimes deliberately break consistency in order to make something stand out, and highlight a particular part of the code*. This indicates a tradeoff between Consistent Design and Clear Layout (see Section 8.2.6). One comment pointed out that *With longer-lived and/or multi-person projects, some inconsistency is going to arise and then you need to decide which to be consistent with*. We agree that deciding between different co-existing consistency schemes is difficult but believe that code that contains multiple consistency approaches is inherently compromised with regard to style. The task is one of making the best of a bad job. For *it can be entirely reasonable to be inconsistent with something that was a bad decision to begin with*, we suggest that making a decision to break consistency because of an objection to the existing scheme is risky as other readers may not agree with your choices and the reduction in consistency will affect them negatively. We have not implemented these suggestions.

Wording One respondent suggested that the third Guideline should be extended to include *should match the existing codebase*. Consistency with the codebase is covered by the first and second Guidelines. We have not implemented the suggestion.

Meaning Several respondents were unclear about the meaning of ‘design’ and ‘design techniques’ in both the name of the principle and in the second and third Guidelines. For the name, we deliberated on how to replace ‘Design’ in a way that addressed both static aspects, for example, naming and choice of data structures, and dynamic aspects, for example, choice of decision and looping constructs. As we could not find a single term that covered both aspects, we renamed ‘Consistent Design’ to ‘Be Consistent’. For the second guideline, we removed the reference to ‘design techniques’ and replaced the guideline with ‘When making decisions about code, be consistent in implementation choices’. For the third guideline, we decided that the inclusion of ‘design and constructs’ was unnecessary as the term ‘code’ was sufficient on its own. The Guideline was altered to ‘Code should adhere to the programming language idiom’.

A.6 Code Style Principle—Non-redundant Content

The Non-redundant Content Principle is:

Principle All elements that are introduced are meaningfully used.

Rationale Redundant content (i.e., unused code and documentation) requires unnecessary mental effort.

Guidelines

Redundant content should be removed to reduce mental effort.

Commented out and unreachable code should be removed.

Statements that do not affect the logical flow of the program (for example, multiple breaks) should be removed.

Students The comment *from the student perspective .. the solution with redundancy is more understandable* implies a misalignment with our intended meaning of ‘redundant’. This Principle relates to content that can be removed without anything being lost, for example, ‘dead’ code and comments that do not relate to the code, for example, out-of-date comments

that have not been removed. The interpretation in this comment is ‘repeated code that has not been extracted into functions’. We acknowledge that the term ‘redundant’ is not the most suitable one for our intention. We have changed the Principle name to ‘No Unused Content’ and have updated the first Guideline to reflect the change.

Exceptions Respondents observed that commented-out code might contain valuable information, for example, to show an alternative approach. We realised that *comments* containing ‘useful’ code should not be removed. To address these, we updated the second Guideline to remove the reference to removing commented-out code. Analysis also caused us to revisit the distinction between the Explanatory Language and Non-redundant Content Principles with respect to comments. We considered (a) comments that repeat what the code says and (b) comments that are out of date. We concluded that (a) is an example of poor commenting and belongs with *Explanatory Language* and (b) is simply unused. This is covered by ‘unused ... documentation’ in the first Guideline.

Wording The viewpoint that the term ‘Non-redundant Content’ was unwieldy has been addressed by the replacement of ‘Non-redundant’ by ‘Unused’.

Meaning Four comments related directly to the difference in meaning between ‘non-redundant’ and ‘unused’ (*‘Redundancy’ can be reduced by referencing a single instance. ‘Unused’ can be reduced by deletion*). We acknowledge that we made a mistake with the term ‘non-redundant’ and have addressed this by the change in naming.

A.7 Code Style Principle—Appropriate Implementation

The Appropriate Implementation Principle is:

Principle Implementation choices are suited to the problem to be solved.

Rationale An implementation that is suited to the problem is easier to change.

Guidelines

Abstractions (for example, data types) and design approaches (for example, control constructs) should be determined by what they are intended to accomplish.

Identifiers and variables should not be reused for different purposes.

Exceptions Two responses noted exceptions to variable reuse *everyone knows when they see ‘i’, if it’s used idiomatically, is a throwaway index variable. Is that a ‘different purpose’ if it’s in a different loop?; If one is working with different classes, there it is hard to see why identifiers and variables should be different..* We agree with this feedback and have added ‘within the same scope’ to the second Guideline.

Scope One respondent did not agree that identifier reuse should be included. We do not agree and believe that ‘appropriateness’ must include a choice of both constructs and variables.

Meaning Several respondents suggested the Principle was too vague (*This principle isn’t as clear as some of the others; What do we exactly mean by abstractions and control approaches?; ‘appropriate’ is very much a matter of ‘opinion’ is not universal.*). We agree that this Principle is less clear and that ‘appropriate’ is difficult to pin down. We have replaced ‘Appropriate’ by ‘Congruent’ in the Principle name, ‘is suited to’ by ‘reflects’ in the Rationale and have reworded the first Guideline. We hope this helps with meaning. For *Sometimes conflicts with Simple Constructs and/or Consistent Design*, we agree that tradeoffs may be necessary and discuss this in Section 8.2.6.

A.8 Code Style Principle—Avoid Repetition

The Avoid Repetition Principle is:

Principle Code duplication is avoided.

Rationale Duplicated code (functions, literals) is difficult to change because there is a risk that not all items are changed.

Guidelines

Duplicated code should be separated out into functions.

Embedded literals that are used multiple times should be abstracted as constants.

Exceptions One comment referred to refactoring out *specific* constants (*initializing sum = 0 or sum = a[0] seems fine*). This covers two separate ideas, *whether* to abstract a constant that has a particular, known meaning and *whether* to include the constant *multiple times*. Decisions about abstracting a constant relate to the Explanatory Language Principle, i.e., the issue is one of not knowing what the constant means. We agree that some constants with a generally accepted meaning and/or use, for example, count initialisers, should not be factored out. We have added ‘that represents a specific concept’ to the second Guideline and to the third Guideline in Explanatory Language (see Section A.2). For repeated code that is expected to diverge (*You might ... duplicate code when you expect that there may be changes later that cause one copy to need to be modified*), we agree that tradeoffs between Principles and other quality characteristics will often be needed (see Section 8.2.6).

Wording We have not implemented a suggestion to remove the phrase *multiple times* from the second Guideline. The purpose behind *whether* a literal is extracted concerns understanding meaning and belongs to Explanatory Language. The purpose behind removing *multiple* instances concerns difficulty in changing and belongs to Avoid Repetition. We discuss the importance of making this distinction in Section 8.2.8.

Meaning Several responses expressed uncertainty about the relationships among Avoid Repetition, Non-redundant Content and Explanatory Language. These each have a different purpose as manifested in the Rationale. Avoid Repetition aims to reduce errors when changing code, Non-redundant Content addresses unnecessary mental effort and Explanatory Language has the goal of understanding the intentions of the author. We discuss the importance of making these distinctions in Section 8.2.8. It was suggested that the name ‘Avoid Repetition’ might be interpreted as advice not to use looping constructs and we changed the name to ‘Avoid Duplication’. One response discussed the need to address other kinds of decomposition, for example, OO. Our intention is that the Principle statement and Rationale are paradigm-agnostic but note the need to explore the CSM for other paradigms (see Section 9).

A.9 Code Style Principle—Modular Structure

The Modular Structure Principle is:

Principle Related code is grouped together and dependencies between groups are minimised.

Rationale Placing related elements together makes code easier to understand. Reducing interconnectedness means that isolated pieces can be more easily understood and can be modified independently.

Guidelines

Modules should be created with clearly defined responsibilities and minimal dependence on other modules.

Functions should aim to have a single responsibility and a minimum number of parameters and side effects.

Code should be organised such that related elements appear together.

The scope of constants and variables should be minimised.

Exceptions The comment *If you want to compute the min, max, average and standard deviation of a LARGE list of numbers, doing each of these separately is pretty inefficient and slow, compared to doing all of them in a single loop.* Illustrates the need for tradeoffs (see Section 8.2.6).

Scope One respondent suggested that *This one includes quite a lot, maybe split up?*. We are not sure if the respondent is referring to cohesion and coupling or to the distinction between higher-level modules (classes and files) and implementation modules (functions). Modularity involves notions of both cohesion and coupling and we believe these must both be included in this Principle. For module type, see ‘Wording’.

Wording Two responses suggested that the terms ‘coupling’ and ‘cohesion’ be introduced. We believe that the definition we have captures what these words mean without introducing unnecessary terminology. One respondent believed that the term ‘single responsibility’ is an OO concept and does not apply directly to functions (*You’re lifting language that already has a meaning in practice, and that’s a very dangerous thing to do to students*). We agree that the use of the term is inadvisable and has been replaced with notion of ‘single task’ in the second Guideline. For *Related code is grouped together is too vague*, we cannot find a better way to express this but have added ‘the most’ to the third Guideline.

Received 27 June 2024; revised 11 November 2024; accepted 16 January 2025