Sprinter: A Didactic Linter for Structured Programming

Francisco Alfredo
Visor.ai Portugal, S.A., Lisbon, Portugal

André L. Santos
ISTAR-IUL, University Institute of Lisbon, Portugal

Nuno Garrido
ISCTE-IUL, IT-IUL, University Institute of Lisbon, Portugal

Abstract

Code linters are tools for detecting improper uses of programming constructs and violations of style issues. Despite that professional linters are available for numerous languages, they are not targeted to introductory programming, given their prescriptive nature that does not take into consideration a didactic viewpoint of learning programming fundamentals. We present Sprinter, a didactic code linter for structured programming supporting Java whose novelty aspects are twofold: (a) providing formative feedback on code with comprehensive explanatory messages (rather then prescriptive); (b) capability of detecting some control-flow issues to a deeper extent than professional linters. We review Sprinter features against popular tools, namely IntelliJ IDEA and Sonarlint.

2012 ACM Subject Classification Social and professional topics → Computer science education; Applied computing → Interactive learning environments

Keywords and phrases structured programming, code quality, code linter

Digital Object Identifier 10.4230/OASIcs.ICPEC.2022.2

Funding This work was partially supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020]. We thank the anonymous reviewers for their valuable feedback.

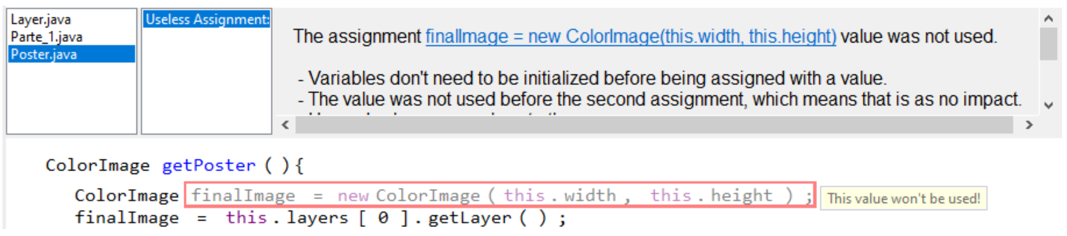
1 Introduction

Code quality has not been widely researched by the programming education community, namely with respect to tooling. Recent studies have identified that quality issues have a significant presence in the code of students [9]. Given the importance of code quality in the software industry, we believe that it is important to bring up the topic early on in the curricula.

From teaching experience, we frequently observe that code quality issues in student code are in part due to their fragile skills with respect to programming constructs. Ruvo et. al. [5] referred to such issues as *semantic style indicators*, and discovered that these are found in the code of students throughout the degree curriculum.

The fact that students develop implementations that work, does not necessarily imply that they fully understand the underlying concepts [8, 14]. Certain aspects of non-optimal student code may have resulted from trial-and-error attempts, combined with unsatisfactory mastery of programming constructs. We speculate that some code quality issues may relate to misconceptions [13], but that investigation is out of the scope of this paper.

In this paper we present Sprinter, a didactic code linter that comprehensively explains code quality issues in terms of what is expressed in the user program, evidencing the problem, rather than the fix (see Figure 1). The perceptions of code quality are not consensual among



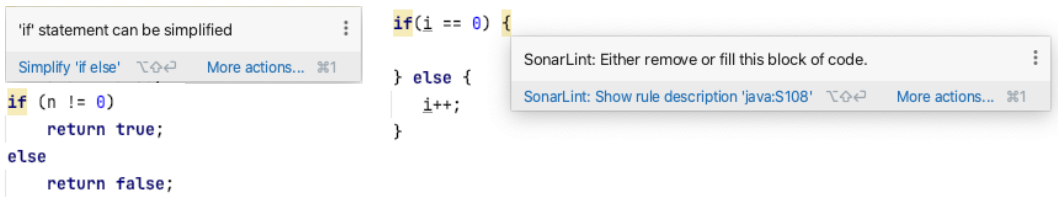
The assignment finalImage = new ColorImage(this.width, this.height) value was not used.

- Variables don't need to be initialized before being assigned with a value.
- The value was not used before the second assignment, which means that is as no impact.

```
ColorImage getPoster ( ){
    ColorImage finalImage  =  new ColorImage ( this . width ,  this . height ) ;
    finalImage  =  this . layers [ 0 ] . getLayer ( ) ;
```

This value won't be used!

Figure 1 Sprinter: student code is analized and annotated with a comprehensive explanation of an encountered issue.


'if' statement can be simplified

```
if (n != 0)
    return true;
else
    return false;
```

```
if(i == 0) {

} else {
  i++;
}
```

SonarLint: Either remove or fill this block of code.

Figure 2 Prescriptive warnings on a code quality issues given by IntelliJ IDEA and Sonarlint.

educators, students, and developers [3]. We focus on quality issues that stem from lack of mastery of elementary programming constructs, as opposed to purely stylistic issues, such as naming and spacing conventions.

Software quality is recommended as a learning goal of Software Engineering curricula [12]. Therefore, didactic tools that raise awareness to code quality issues are an advantage towards this aim. In our contribution we address quality issues within the scope of optimal usage of structured programming constructs, which fundamentally involve sequence, branching, and looping. We address characteristics of programs that in isolation or simultaneously: (a) lead to useless computations, (b) exhibit redundancy in their instructions (duplication), or (c) are unnecessarily verbose. We aim at providing code quality feedback targeting novice programmers, differing from professional linters in terms of message format and improvements in issue detection.

Sprinter is a proof of concept for Java. However, the issues are not Java-specific, and our approach is applicable to languages that have similar structured programming constructs. So far, we have developed 14 detectors for structured programming issues, mostly relying on control-flow analysis. In addition to presenting the issues centered on the problem rather than the fix, our tool detects a few issues that some professional linters are not able to detect. A tool such as Sprinter could be useful in programming courses, given that it can raise awareness of knowledge gaps and deliver hints for students to improve their skills.

2 Related work

Professional code linters may be used in teaching, as they typically provide plugins for widely-used IDEs (e.g., Sonarlint[1]) or are integrated in the IDEs themselves (e.g., IntelliJ[2]). These industrial linters have a *prescriptive* nature in their interaction, since they simply tell *what to do* to fix an issue, typically with an accompanying *quick fix* option for performing the

[1] https://www.sonarlint.org
[2] https://www.jetbrains.com/idea

operation automatically. For didactic purposes, we believe that a tool should explain *why* some aspect of the program is not as good as it could be, elucidating the user as much as possible, while not providing a “blind fix” that can be applied without understanding. Our goal is that students can understand an issue, and ideally, fix it autonomously without the aid of automatic features. Otherwise, one may fix without understanding, which we argue is not ideal for the learning process.

The code linter concept has been applied to early stages of programming learning. LitterBox [6] is a linter for Scratch programs that is capable of providing hints to users based on a catalog of patterns, which are used to analyze the abstract syntax tree. One of the main motivations for this approach is aligned with ours. The fact that learners produce working code with correct outputs does not imply that they are using programming constructs in the best way and have no misconceptions regarding their application.

Keuning et al. [9] have analyzed a large amount of Java code from the BlackBox dataset [4], finding that quality issues have a significant presence in student code. Further, they observed that some quality issues tend to remain as a code file evolves. The same authors have conducted a study with programming teachers [10] in order to investigate the type of quality feedback they would provide to students. They observed that teacher feedback consists of aspects pertaining to reducing algorithmic complexity that is not covered by professional tools, such as simplifications of `if-else` statements. We address several of those issues in Sprinter.

Keuning et al. [11] also investigated if students were able to address quality issues using a refactoring tutor, where instructors develop exercises that start with working code with quality issues, and further provide hints for students to fix, while checking progress when correct steps are taken. Sprinter is similar to such a tutor, but it supports detection of quality issues in arbitrary student code. Hence, students become aware of quality issues by means of their own code, rather than through instructor-designed cases.

FrenchPress [2] is a diagnosing system for Java programs that focuses on issues related to object-orientation and use of variables. Sprinter has only a small overlap with FrenchPress with respect to boolean expressions. Our focus is on structured programming constructs, and nearly to practically all the issues we detect are not handled by FrenchPress. Hence, the diagnoses of both tools could be combined into a broader tool.

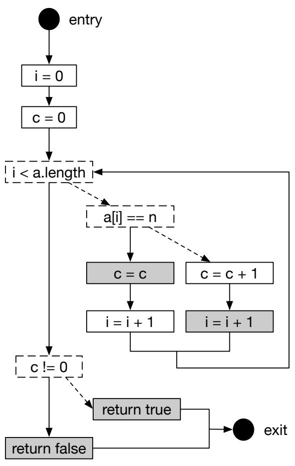
The CompareCFG tool [7] provides students with a control-flow graph (CFG) for their submitted solution, side by side with a CFG of another submitted solution that is less complex. The goal is that students can improve their code autonomously by comparing the solutions and reading additional actionable feedback provided by the tool. This approach is similar to ours, as we also make use of a CFG to detect issues and derive actionable feedback, but we do not present it to students.

Hyperstyle [1] is a tool for automated evaluation of code quality that relies on reusing the functionality provided by professional linters. The messages that explain the issues are not those provided by linters, but rather custom designed by the tool. However, no details are available so far regarding how issues are presented. Hyperstyle supports several languages, but with respect to Java, it relies on the linters Checkstyle[3] and PMD[4]. The latter address mostly syntactic stylistic issues, whereas issues related to control-flow are not detected. In our work we focus on issues related control-flow and structured programming constructs.

[3] `https://checkstyle.sourceforge.io`
[4] `https://pmd.github.io`

```
boolean contains(int[] a,
    int n) {
  int i = 0;
  int c = 0;
  while(i < a.length) {
    if(a[i] == n) {
      c = c + 1;
      i = i + 1;
    }
    else {
      c = c;
      i = i + 1;
    }
  }
  if(c != 0)
    return true;
  else
    return false;
}
```

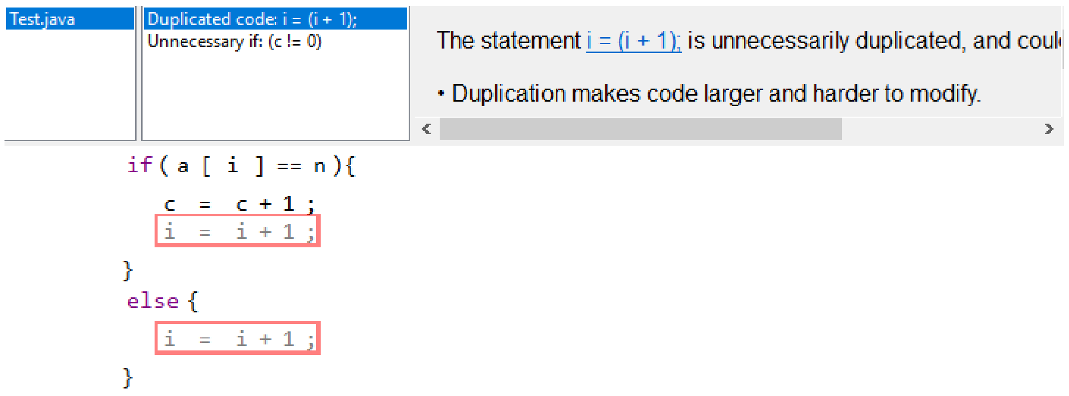
(a) Code example with quality issues.



(b) Control-Flow Graph of (a).

```
boolean contains(int[] a,
    int n) {
  int i = 0;
  int c = 0;
  while(i < a.length) {
    if(a[i] == n)
      c = c + 1;
    i = i + 1;
  }
  return c != 0;
}
```

(c) Version of (a) with improved quality.



Test.java

Duplicated code: i = (i + 1);
Unnecessary if: (c != 0)

The statement i = (i + 1); is unnecessarily duplicated, and coul

• Duplication makes code larger and harder to modify.

```
if(a[i] == n){
   c = c + 1;
   i = i + 1;
}
else{
   i = i + 1;
}
```

(d) Presentation of the quality issues in Sprinter (when `c=c` was already addressed).

Figure 3 Quality issue detection based on control-flow analysis.

3 Control-flow analysis

With the exception of the more trivial cases, most of the issues detected by Sprinter rely on control-flow analysis. We derive a CFG for each procedure (i.e., method, using Java's terminology) of the code under analysis. A CFG models a procedure with an *entry* and *exit* point, statements (nodes), and transitions (edges). Each node is either a statement, or a branching point that determines the following statement according to the evaluation of an expression (control structures). The detection of code issues is performed by querying the CFG for "anomalies".

Figure 3a presents a code example which, despite its contrived appearance, combines several aspects reported in previous studies [5, 9] that any programming education teacher with some experience has seen. For illustration purposes, we combine three code issues in a single example. Notice that in addition to the code issues, algorithmic-wise, the solution is also not optimal (the array iteration does not have to be complete, and no counter is strictly necessary). However, our approach is not concerned with algorithmic strategies, but rather in how those are expressed. Figure 3b presents a CFG that models the code of Figure 3a. The gray instructions are "superfluous" and could be removed through the process of reaching the equivalent solution presented in Figure 3c.

The most trivial issue in the code is the self assignment `c=c`. Given its redundancy, it could be eliminated without any elaborate analysis. Further, by analyzing branches to check that all starting/ending have the same statements (`i=i+1`), we detect that they could be factored out from their branch. As so, the `else` branch becomes empty, and in turn, may be removed as well. Finally, the last `if` is an identity map of the expression `c!=0` to the `return` expression, and hence, it may be simplified to `return c!=0`.

Another aspect that we rely on for detecting code issues is the nature of procedures, namely if they are either *pure functions* without side-effects, or *procedures* that modify state. We determine this through static analysis. When there are no references passed to a procedure, or when all the referenced arrays/objects are not modified nor passed to a procedure, we classify it as pure function. We also distinguish between constant-time and non-constant-time function, by analyzing the function body.

4 Sprinter

We developed the Sprinterprototype supporting Java. Although it is an object-oriented language, in this work we are only focusing on structured programming constructs, which are available across many programming languages with equivalent or similar semantics (e.g., C, Python, Matlab, R). The tool is currently standalone, but it would certainly make sense to integrate it in an IDE in the future (e.g., Eclipse, IntelliJ, VS Code). We also aim at

Sprinter takes as input Java files, which are checked against issue detectors. Each encountered issue is presented to the user in isolation, using annotations in the code (see Figure 3d). We highlight the parts of the code that are involved in the issue and we may attach a small warning sentence about the problem (see Figure 1). In a separate panel, we provide an accompanying explanation of the issue and related concerns. The text may hold links to the code to aid the user in relating the explanation to the code. We do not provide any quick fix options to solve the issues automatically.

So far, we managed to successfully implement detectors for 14 issues, of which we present the ten-most significant in Table 1. We have omitted the more trivial cases such as self-assigning a variable or "tautology if-guards" consisting of the literal `true`. Each case is illustrated with a sketch-example, and we indicate if IntelliJ (version 2022.1.1) and Sonarlint (version 6.4.3) are capable of fully or partially detecting the issue. The former can be considered to be one of the most advanced IDEs for Java, whereas the the latter is a leading professional linter. Notice that in order to have the coverage of issue detection of Sprinter one has to use a combination of two widely used professional linters. Using multiple (professional) tools in educational settings creates undesired overhead.

In addition to the form in which code issues are presented, Sprinter detects some issues that are not flagged by some industry-strength tools. Notice that there are issues not (fully) detected by either IntelliJ or Sonarlint. We are able to achieve this in part because we reduce the scope of code analysis, currently by not delving into Java's library classes. That is, the classification of pure functions is not performed in these cases, as required by some detectors.

The issue of Magic Numbers is prone to some subjectivity. We compromise by flagging cases when the same literal is found twice or more within the same procedure (excluding literals 0, 1, and 2). However, the solution is not perfect, and achieving one is not trivial. A same number may refer to different unrelated things, and that is hard to determine with precision. On the other hand, the often-used 0, 1, and 2, may also be used in situations where a constant would make sense.

Table 1 Code issues detected by Sprinter with illustrative examples. We mark the issues for which there is an equivalent detection available in IntelliJ IDEA (IJ) and Sonarlint (SL). Full support: ●; partial support: ◐; [a] Does not take into account semantically equivalent duplication (see code example); [b] Empty `if` is signaled, but `else` is not take into consideration in the explanations.

Name	Description	Example (Java)	IJ	SL
Useless Assignment	Assignment of a value that is not used.	`int[] array = new int[100];` `array = newRandomArray(100);`	●	●
Useless Call	A call to a function that has no side-effects without making use of the returned value.	`copy(array);`	●	
Useless Return	Return at the end of `void` methods.	`void doSomething() {` `...` `return;` `}`	●	
Identity Return	Mapping the evaluation to separate return statements.	`if(bool) return true;` `else return false;`	●	●
Redundant Equality	Comparing a boolean expression to a boolean literal.	`if(boolExpression == true) {` `...` `}`	●	●
Redundant Call	A call to a non-constant pure function with the same arguments.	`int m = max(list);` `list.remove(max(list));`		
Redundant Guard	A guard that is checking a condition that is known to be true (given the enclosing control structure).	`while(i > 0) {` `if(i > 0) { ... }` `}`	●	
Duplication in Branches	Common behavior in alternative branches that could be factored out.	`if(v[i] > 0) {` `c++;` `i++;` `} else {` `i+=1;` `}`	◐[a]	
Counter Guard Branching	Empty block in `if` with desired behavior in `else`.	`if(guard) {` `} else {` `// do something` `}`	◐[b]	◐[b]
Magic Number	Numeric literal that is used without an explanation.	`if(c > 255)` `c = 255;`		●

5 Conclusions

We presented a novel form of explaining code quality issues related to structured programming constructs embodied in the Sprinter tool. Some of these issues are not, or are only partially addressed by professional tools. As future work, we plan to address other issues, such as code duplication at the expression level, as well as other issues related to branching.

A tool such as Sprinter, if properly integrated in the programming practice workflow, could be beneficial to raise awareness to code quality, while helping students to improve their code autonomously. Tool support becomes even more relevant in the context of large-scale course where human tutoring is scarce or not even available. As future work, we plan to carry out a user study with programming beginners to investigate how they can cope with the issues raised by Sprinter.

References

1 Anastasiia Birillo, Ilya Vlasov, Artyom Burylov, Vitalii Selishchev, Artyom Goncharov, Elena Tikhomirova, Nikolay Vyahhi, and Timofey Bryksin. Hyperstyle: A tool for assessing the code quality of solutions to programming assignments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, pages 307–313, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3478431.3499294`.

2 Hannah Blau and J. Eliot B. Moss. Frenchpress gives students automated feedback on java program flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pages 15–20, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2729094.2742622`.

3 Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. "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, pages 70–85, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3174781.3174785`.

4 Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 223–228, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2538862.2538924`.

5 Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. Understanding semantic style by analysing student code. In *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18, pages 73–82, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3160489.3160500`.

6 Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. Litterbox: A linter for scratch programs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 183–188, 2021. `doi:10.1109/ICSE-SEET52601.2021.00028`.

7 Lucy Jiang, Robert Rewcastle, Paul Denny, and Ewan Tempero. Comparecfg: Providing visual feedback on code quality using control flow graphs. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, pages 493–499, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3341525.3387362`.

8 Cazembe Kennedy and Eileen T. Kraemer. Qualitative observations of student reasoning: Coding in the wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, pages 224–230, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3304221.3319751`.

9 Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 110–115, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3059009.3059061`.

10 Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 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, pages 119–125, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3304221.3319780`.

11 Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Student refactoring behaviour in a programming tutor. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, Koli Calling '20, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3428029.3428043`.

12 T. C. Lethbridge, R. J. Leblanc Jr, A. E. Kelley Sobel, T. B. Hilburn, and J. L. Diaz-Herrera. Se2004: Recommendations for undergraduate software engineering curricula. *IEEE Software*, 23(6):19–25, 2006. `doi:10.1109/MS.2006.171`.

13 Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, 18(1), October 2017. `doi:10.1145/3077618`.

14 Jean Salac and Diana Franklin. If they build it, will they understand it? exploring the relationship between student code and performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, pages 473–479, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3341525.3387379`.