Understanding semantic style by analysing student code

Conference Paper · January 2018

DOI: 10.1145/3160489.3160500

5 authors, including:

Andrew Luxton-Reilly
University of Auckland
144 PUBLICATIONS **3,999** CITATIONS

SEE PROFILE

Gerard Rowe
University of Auckland
56 PUBLICATIONS **786** CITATIONS

SEE PROFILE

Understanding Semantic Style by Analysing Student Code

Giuseppe De Ruvo
Dep. of Electrical and Computer Engineering, The University of Auckland, Auckland, New Zealand
g.deruvo@auckland.ac.nz

Ewan Tempero
Andrew Luxton-Reilly
Dep. of Computer Science, The University of Auckland, Auckland, New Zealand
e.tempero@auckland.ac.nz
a.luxton-reilly@auckland.ac.nz

Gerard B. Rowe
Nasser Giacaman
Dep. of Electrical and Computer Engineering, The University of Auckland, Auckland, New Zealand
gb.rowe@auckland.ac.nz
n.giacaman@auckland.ac.nz

ABSTRACT

Good coding style is recognised by the software engineering profession as being important, and this is reflected in the standard computing curricula. Feedback on some aspects of coding style is now commonly provided by IDEs and by tools such as Checkstyle, but this feedback focuses on coding standards that are largely based on syntax. However, some aspects of coding style relate to the semantics of code — of the many ways to achieve some functionality, some are preferred because they are simpler, yet students struggle to create them. In this paper, we introduce the concept of semantic style, and in particular semantic style indicators that may be manifestations of poor knowledge of some programming concepts. We describe 16 semantic style indicators and demonstrate their prevalence in almost 19,000 code samples submitted by over 900 novice students. Half the students submitted code exhibiting two or more of these indicators, demonstrating the potential value to learn by providing feedback on semantic style. We also find many indicators are present in the code of students attending their fourth year of a highly competitive Software Engineering programme, demonstrating the need for more attention to teaching of semantic style issues.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; **Information science education**; *Student assessment*;

KEYWORDS

Semantic style, coding standards, software quality, novice programmers, formative feedback, automated feedback

1 INTRODUCTION

As any experienced software developer knows, there is more to engineering software than just getting the code to work. A significant part of the lifetime cost of a software system is incurred after its first deployment, that is, as maintenance. This means that there is a strong incentive to identify and remove problems in code that may incur future costs. Programming style guidelines have been developed to help reduce potential problems. These guidelines typically apply to the organisation of the structure of the code, that is, the use of syntax. We refer to such guidelines as applying to *syntactic style*. Such recommendations do not change the choice or order of statements, but do change how they are presented, such as how they are laid out (e.g. indentation, spacing, use of line breaks) or what identifiers they use (e.g. naming conventions).

In this paper, we are interested in addressing the learning and assessment of *semantic style*. This interest stems from our observation that some potential problems with learning to program manifest as *semantic* issues. That is, the code may be well-structured and correct, but have features that an experienced programmer would typically avoid. An example of a possible semantic style issue is shown in Figure 1, where a student likely does not fully know how to use Boolean variables in `if` statements.

```
if(a == true)
  return true;
else
  return false;
```

Figure 1: Example of a semantic style issue

It is considered important to include the presentation and organisation of program code in lessons on code quality. In the Computer Science Curriculum Guidelines [7], Software Development Fundamentals (SDF) include Learning Outcomes (LO) expecting students not only to apply consistent programming style standards to improve readability and maintainability of their own code (SDF LO#12), but also to examine whether code written by others also meets those standards (SDF LO#11). Similarly for the Software Engineering Curriculum Guidelines [8], Software Verification and Validation (VAV) LOs also emphasise the importance of personal reviews (VAV.rev.1) and peer reviews (VAV.rev.2) in their role of contributing to Software Quality [32, 47].

Reviewing and assessing code quality is a challenge, in part due to the high number of students typically enrolled in introductory programming courses, and in part due to the fact that current processes for doing so are largely manual. In particular, it is difficult to provide constructive *and* timely feedback. In some courses, students are encouraged to follow specific conventions without formative feedback on their code style, particularly when automated assessment tools are used to determine code correctness. Furthermore, writing good software is more than following known conventions.

ACE 2018, January 30-February 2, 2018, Brisbane, QLD, Australia

ACM ISBN 978-1-4503-6340-2/18/01…$15.00
https://doi.org/10.1145/3160489.3160500

For novice programmers to truly develop a deeper understanding and appreciation of writing quality code, less teaching and more personalised feedback is likely to be more constructive [45]. Ultimately, timely formative feedback is essential in developing students into self-regulated learners [35, 36].

Providing immediate feedback showing how the code in Figure 1 may be simplified (to just `return a;`) may help the student improve their knowledge of the semantics of their code. Being able to automatically detect such issues would help teachers to efficiently and quickly identify students in need of guidance in this area [38].

The questions we address in this paper are:

RQ1 Are there common semantic style issues?

RQ2 Can semantic style issues be automatically identified and at a reasonable cost?

Novice programmers typically write code that is more complex than necessary. It is possible that there are so many different potential semantic issues that it might be difficult to identify a subset that will be useful for teaching and learning. RQ1 addresses this concern. Analysing semantic behaviour of code is sometimes very difficult and time consuming. If that is the case with the kinds of semantic style issues we are interested in, that would also make it difficult to provide fast feedback, hence RQ2.

We demonstrate that both questions can be answered positively. We do so by mining a large collection of student submissions. These submissions include both single-method coding exercises produced by CS1/CS2 students, and larger assignments generated by students of a software design course held in the fourth year of a highly competitive Bachelor in Software Engineering. From this, we have identified 16 semantic style issues that are common to many submissions. We have developed a prototype tool for detecting these issues. The tool only requires the Abstract Syntax Tree (AST) representation of the code, and so is very efficient, and should be able to be easily incorporated into any IDE.

The remainder of the paper is organised as follows. In Section 2, we discuss the relevant related work. Section 3 describes the study we performed to investigate the usefulness of the semantic style indicator concept, as well as the tool we developed to detect the indicators. Section 4 presents the semantic style indicators we have identified. We present the results of our study in Section 5, and discuss their significance in Section 6. Finally we present our conclusions and discuss future work in Section 7.

2 RELATED WORK

The concept of semantic style is related to "code smells". According to Fowler, the term "code smell," coined by Beck [22], is "a surface indication that usually corresponds to a deeper problem in the system [21]". Fowler and Beck use this concept as a means to identify when refactoring should be considered. A code smell applies at the design level, that is, situations involving multiple classes (or when there should be multiple classes, such as the **Large Class** code smell). In contrast, what we mean by semantic style applies at the statement level.

Research specifically on code smells has featured tools to detect and report smells (e.g. [33, 43]). There has been research on the relationship between the presence of smells and other quality issues such as fault-proneness [23] and maintainability [41, 48]. Recently Tufano et al. performed an empirical study to better understand what leads to smells being introduced into code [44]. One result was that code smells are often introduced at the creation of code, rather than, as we might expect, a consequence of the evolution of the software. This suggests that early detection of any code problems, including semantic style issues, would be of value.

The existence of a code smell does not mean that a refactoring must be done, but is just an indication that maybe something is not right, and so should be examined more closely. The problem that caused the smell may have been due to the developers' understanding of the design being incomplete, or more generally having only a superficial understanding of what constitutes a good design, that is, they lack good mental models of how to create good designs. They may otherwise be expert at writing code at the statement level.

Very similar to code smells are anti-patterns, i.e. poor design solutions affecting software quality. In fact, they have been discussed because they also slow down the comprehension and maintainability of software systems [25, 37]. Poor choice of names is a linguistic ant-pattern, and may impair program understanding [12].

Unlike expert programmers, novice programmers lack detailed mental models and so deal with their programs one line at a time [30, 39]. With such a surface-learning approach, novice programmers will primarily focus on program functionality and be tempted to neglect good coding practices [40]. Indeed, novice programmers are more likely to make poor choices despite the frequent advocation of good programming style [31]. Timely formative feedback [9, 17] helps avoid the manifestation of bad long-term habits [11].

Many tools providing feedback on student programming exercises have been reviewed [10, 29]. Despite the large collection of such tools, feedback tends to focus dominantly on dynamic assessment; this includes functionality against provided requirements, efficiency of program execution or test coverage [10].

More relevant to semantic style are tools that perform static assessment of the code. Many modern compilers warn programmers of dead code and unused variables, but do little regarding (the somewhat more subjective) adherence to good coding standards. Tools that most closely resemble feedback on coding style provide feedback on typographic aspects such as commenting and indentation [10, 24], or require human involvement for the feedback stage [26].

Tools such as Checkstyle [1] and Sonar [6] are used to ensure code in a project conforms to a defined coding standard style. While this helps detect style violations, the goal is on improving style consistency (for the project's benefit) rather than to help the user improve their programming understanding. Tools such as PMD [5], FindBugs [3], Google Error Prone [2] and Facebook Infer [4] are useful in identifying flaws that could manifest themselves as bugs [11]. FindBugs [3] and PMD [5] have been proven to be underused [20], and so confusing (even for professionals) as to require the need of an intuitive visualization to compare the results [16]. Thus, these tools are most likely not suitable for students and "may cause confusion with students" as advised by Keuning et al. [28]. PMD [5] has been also used to investigate "Static Analysis Errors in Student Java Programs" [20]. We are, instead, interested in (semantic) style issues in code that have no other flaws.

A recent paper [28] investigated "Code Quality Issues in Student Programs" using the Blackbox dataset [15]. Keuning et al. [28]

analysed four weeks of programs submitted in the academic year 2014-2015. Instead, we consider only final submissions of different years and courses, i.e. there are no snapshots and all the code submitted is complete (See Section 5). Furthermore, we know exactly what the students' assignments were and which kind of task each student needed to accomplish, whilst Keuning et al. [28] gave no information about the nature of the assignments. Finally, all the code in our datasets was submitted by students and there is no threat of analysing code written by instructors or more experienced programmers.

3 METHODOLOGY

Our eventual goal is to improve coding practices of novice programmers by providing relevant feedback on quality as soon as possible. Many programming texts provide advice regarding what good coding practices are, but it is generally not specific enough to be usable in a tool. Instead, we have chosen to focus on sequences of statements that suggest the novice programmer may have an incomplete understanding of how to best express their solution using the programming language [14].

We identified candidate semantic style problems by examining the solutions provided by students to a set of exercises used in an introductory programming course at The University of Auckland. The exercises required students to implement a single method to satisfy the required functionality, that is, a typical early assignment for such a course. The students used CodeWrite [18], a programming drill-and-practice web-browser based system. This presents the requirements for the method, and a text box in which the student writes her solution. On submission, the student's submission is compiled and run against a suite of test cases. If the submission does not compile, or does not pass one of the tests, then feedback is provided. Once the submission passes all tests, the exercise is complete.

The dataset we used in the initial stage has 10 exercises, with successful submissions from between 130 to 180 students each. We initially manually examined solutions to one exercise to find candidate style problems. Once we had a candidate, we then developed our tool to determine whether the candidate appeared in any other submissions from any other exercise. We repeated this process as often as necessary to get a set of interesting candidates. We describe the semantic style indicators we identified in the next section.

Our detector works with the Abstract Syntax Tree (AST) representation of the submission. As many Integrated Development Environments (IDE) SDKs give access to the AST, this provides the opportunity to integrate the detection and feedback into the IDE, such as by means of plugins. It also means that detection is very quick, as the analysis is performed on a fairly simple representation.

We employed the Eclipse Java Development Tools project (JDT) [19] as our implementation platform. The JDT project provides tools to create Java applications and also APIs to access and manipulate Java source code. It allows convenient access to the AST of the Java source code and is completely integrated into the Eclipse IDE. We implemented our tool using the JDT core package accessing directly the Java AST for finding "semantic style issues" (see Section 4). A custom Visitor [27] was developed for each of the semantic style issues we identified, to traverse the AST and report occurrences of the particular style issue.

Having identified a set of semantic style indicators and a tool for detecting them, we analysed the complete original dataset, to confirm the tool's accuracy. We then analysed 3 other datasets — two other, larger, CS1 datasets and one from a design class. The results of these analyses are in Section 5.

4 THE INDICATORS OF SEMANTIC STYLE

We are interested in surface indications in code that suggest the programmer lacks some knowledge about one or more aspects of programming. Such indicators can be found even in a single line of code Since all the code snippets compile and are functionally correct, it is not a syntax issue or something for static analysis tools as used by experienced developers [13], e.g. the aforementioned CheckStyle, or FindBugs, PMD, Sonar etc. It is potentially a point of misunderstanding, and hopefully a signal for students (and instructors) to investigate where students need to reconsider their coding style.

A *semantic style indicator* is a pattern of a short sequence of statements that in some circumstances could be considered sub-optimal. Sub-optimal might mean that the sequence is longer than it need be, but could mean there are alternatives that are simpler or easier to understand. As with syntatic style, there may be circumstances when the indicator is the preferred choice due to the context in which it appears, but usually an alternative would be preferred

Following the methodology outlined in the previous section, we defined 16 indicators of semantic style, which we describe below. The examples shown come from the exercise submissions in the dataset called "CS1-intro-10ex" — the dataset used to identify candidate style indicators. In each of these descriptions, we suggest what might have led to the advent of the particular indicator. We may be wrong in some cases; it could be that the student was merely in a hurry to get the exercise complete and chose the most expedient solution. Such uncertainty is inevitable with *a posteriori* analysis. If the choice of coding style was in fact deliberate, the feedback will merely confirm what the programmer already knows, so there is no harm done. If however there was confusion (or lack of knowledge), then this is a good opportunity to improve learning.

4.1 Unnecessary IF/ELSE-IF (UIF)

Two `IF`s or combination of `ELSE-IF` checking the same variable, but one of the conditions is the negation of the other. An example of this indicator is shown in Figure 2. The programmer appears uncertain as to how `IF` and `ELSE-IF` operate. Feedback can be provided showing that, in this case, the second `IF` (and its condition) can be eliminated.

```
if(positivesOnly == false) {
  ...
} else if(positivesOnly == true) {
  ...
}
```

Figure 2: Example of Unnecessary IF/ELSE-IF (UIF)

4.2 IF Return Condition (IFRC)

Figure 3 show an IF−ELSE statement which, after checking a condition, returns true (IF−body) or false (ELSE−body). There appears to be uncertainty regarding what is allowed for the expression in the return statement. Feedback could show how the Boolean expression in the IF condition can be used in the return statement.

```
if(product % 2 == 0) {
  return true;
} else {
  return false
}
```

Figure 3: Example of IF Return Condition (IFRC)

4.3 IF Return True (IFRT)

This is similar to the IFRC indicator, but the second return statement is outside the IF and without the ELSE−statement (Figure 4). This may indicate uncertainty regarding the use of ELSE. While this seems a mere variation on the IFRC indicator, it could be due to a different aspect of knowledge; so, we treat it separately until we have enough evidence to resolve the question (see Section 6).

```
if(a * b % 2 == 0) {
  return true;
}
return false;
```

Figure 4: Example of IF Return True (IFRT)

4.4 IF Assign Value (IFASNV)

In this indicator, a value/expression is assigned to a variable in one of multiple IF/ELSE−IF/ELSE statements, and that variable is immediately returned as shown in Figure 5. This may be a consequence of following advice to only have one return per method. But it also may indicate confusion on the return of values/expressions within IF−statement bodies. Instead of using a temporary variable, feedback could suggest to just return the value.

```
if(increment) {
  numberOfValues++;
} else {
  numberOfValues--;
}
return numberOfValues;
```

Figure 5: Example of IF Assign Value (IFASNV)

4.5 IF Assign Boolean (IFASNB)

This indicator assigns a Boolean value to a variable, and it then returns that variable as shown in Figure 6. It is a combination of IFASNV (doing away with the variable) combined with IFRC (doing away with the IF−ELSE altogether).

```
if(divis == 0) {
  divisor = true;
} else {
  divisor = false;
}
return divisor;
```

Figure 6: Example of IF Assign Boolean (IFASNB)

4.6 Empty IF Body (EIFB)

This indicator includes an IF−ELSE statement with an empty body for the IF section, as shown in Figure 7. This may suggest a belief that IF statements must always have both clauses, or a confusion regarding the Boolean expression (e.g. the student does not know how to negate the clause).

```
if(values[i]%2 == 0) {
} else {
  j++;
}
```

Figure 7: Example of Empty IF Body (EIFB)

4.7 Empty ELSE Body (EEB)

This is similar to the previous Empty IF Body indicator, but this time the ELSE section contains the empty body as shown in Figure 8. This could suggest a lack of knowledge regarding the use of IF/ELSE or, like the previous indicator, a possible belief that IF statements must always have both clauses.

```
if(values[i] % 2 != 0) {
  counter++;
} else {
}
```

Figure 8: Example of Empty ELSE Body (EEB)

4.8 Unnecessary Nesting IF (UNIF)

This includes two (or more) nested IF statements, where the nested IFs contain nothing other than inner IF bodies. An example of such an indicator is shown in Figure 9. In this case, the student may lack understanding of logic connectors (e.g. AND in this case). Alternatively, it could suggest the student is thinking about the algorithm "step by step" — maybe just reading the assignment specifications, without concentrating on codifying this more elegantly.

```
if(positivesOnly) {
  if (values[i] >0) {
    sum += values[i];
  }
}
```

Figure 9: Example of Unnecessary Nesting IF (UNIF)

4.9 Confusing Else (CE)

This indicator includes an ELSE body with *nested* IF (and/or IF-ELSE) statements within it, rather than simply using a flat IF/ELSE-IF/ELSE approach. An example is shown in Figure 10. The lack of knowledge may be in regard to the use of logical connectors, e.g. AND, or the student does not know how to organise the control flow. In the latter case, we should teach that it is important to distinguish the different control flows — especially for future maintenance purposes.

```
if(age < 20) {
  yearsOver20 = 0;
} else {
  if(age > 40) {
    yearsOver20 = 20;
  } else {
    yearsOver20 = age - 20;
  }
}
```

Figure 10: Example of Confusing Else (CE)

4.10 Unnecessary Else (UE)

This includes an ELSE body with one statement that is duplicated as the last statement in the corresponding IF-body. An example of UE is shown in Figure 11. The shared statement can be moved outside the IF-ELSE, therefore eliminating the ELSE entirely. The student may struggle with "factorising" common statement (possibly struggling with control-flow understanding), or thinks (as in EIF and EEB) that an IF is always followed by its ELSE clause.

```
if(values[i] > 0) {
  sum+=values[i];
  i++;
} else {
  i++;
}
```

Figure 11: Example of Unnecessary Else (UE)

4.11 Useless Duplication in IF/ELSE (UDIE)

This is a more general case of the Unnecessary Else indicator, since the common statement forms a proper subset of the ELSE body (i.e. the ELSE is still necessary since it contains additional statements not inside the IF-body). An example of UDIE is shown in Figure 12. This is likely to arise from struggling to "factorise" the common statement (similar to UE), or simply not noticing the duplication.

```
if(value%2 == 0) {
  numEvens++;
  count++;
} else {
  numOdds++;
  count++;
}
```

Figure 12: Example of Useless Duplication in IF/ELSE (UDIE)

4.12 Iterated Useless Duplication in IF/ELSE (IUDIE)

Extending UDIE, this indicator now reports *two or more* duplicated statements between the IF and ELSE bodies. We distinguish this from the UDIE's "only one statement duplicated", as it could lead to different feedback/refactoring suggestions. An example is shown in Figure 13 with an IUDIE of size 3 (a duplication of 3 statements). Such an indicator is a strong hint that the student is undervaluing software qualities such as conciseness and code reuse, although it is still possible the student is simply not confident in factorising the common statements outside of the IF-ELSE statement.

```
if (age <= 20) {
  int payRate = 15;
  int overtimeRate = payRate * 2;
  int pay = (normalHours * payRate) +
     (overtimeHours * overtimeRate);
  return pay;
} else {
  int payRate = 35;
  int overtimeRate = payRate * 2;
  int pay = (normalHours * payRate) +
     (overtimeHours * overtimeRate);
  return pay;
}
```

Figure 13: Example of Iterated Useless Duplication in IF/ELSE (IUDIE)

4.13 Useless Declaration/Assignment Division (UDAD)

Here, a variable declaration statement is *immediately* followed by an assignment to that variable, i.e. the declaration is not separated with the intention of the variable being used across different blocks or scopes. An example of UDAD is shown in Figure 14. Regardless of whether the student is aware that variables may be assigned values at the time of declaration (or there is a wider issue of how to use variables in general), it is still worthwhile informing them of the good practice to initialise values at the time of their declaration [42].

```
int length;
length = word.length();
```

Figure 14: Example of Useless Declaration/Assignment Division (UDAD)

4.14 Unnecessary Cast (UC)

In this indicator, a cast is added even when it is not necessary (either because it is automatically done by the language, or unnecessary altogether). An example of UC is shown in Figure 15, where the `items` variable is an `integer` type. In this case, the student may not know how the arithmetic operations work or she was just unsure and added the cast anyway. A suggestion of eliminating the cast with an explanation on why would help the student to clarify the concept.

```
int fullContNeeded = (int)(items/10);
```

Figure 15: Example of Unnecessary Cast (UC)

4.15 Known Indicators

Two of the 16 semantic style indicators we identified from the student submissions are already known to the software engineering community as they are detected by many IDEs. Because they were identified from our methodology, we include them below for completeness. We also want to investigate whether and how many are present in student code (in addition to the other 14 indicators). Recall that students may have developed the code without an IDE, or an IDE that does not report them, so their presence may indicate lack of knowledge regarding usage of variables.

```
sum = sum;
```

Figure 16: Example of Selfish Variable (SV)

4.15.1 Selfish Variable (SV). This denotes a variable assigned to itself, as shown in Figure 16. Feedback given back to students needs to help them realise of its redundancy, hopefully encouraging them to reflect on their coding goal.

4.15.2 Lonely Variable (LV). A variable declared and never used, as shown in Figure 17. The feedback would be to point out that this is redundant.

```
int product = a * b;
if (a * b % 2 == 0) {
  return true;
}
return false;
```

Figure 17: Example of Lonely Variable (LV)

5 EXPERIMENTAL RESULTS

We started by analysing 1593 solutions to 10 simple exercises produced by students in a typical CS1 course, the dataset we call "CS1-intro-10ex". These exercises are:

- **replaceCharAtPos**: replace a character in a word at the given position;
- **productIsEven**: given two numbers, return whether the product is even or not;
- **swapEnds**: swap first and last character of a given word;
- **containersNeeded**: return how many containers we need for a given number of items. Each container has a capacity of 10;
- **weeklyPay**: calculate the weekly pay of an employee with an age-based pay-scale. The inputs to the exercise are the number of normal hours worked, the number of overtime hours worked and the age of the worker;
- **countOdds**: calculate the number of elements in an array that are odd;
- **posOfValInArray**: find the position of an input value in a given array;
- **reverse**: return an array of integers in which the elements are in the reverse order to the original input array;
- **magicNumbers**: check whether two given numbers satisfy the condition of being "magic": different numbers containing the same number of digits, and the smaller number is a divisor/factor of the larger number;
- **sumValues**: sum either all of the elements or only the positive elements in the array depending on the value of a Boolean input.

As stated in Section 4, we employed the CS1-intro-10ex dataset to develop our methodology and we manually validated it exhaustively. Following this, we ran our tool against two other datasets: CS1-var-2010 and CS1-var-2011 (or CS1-var to refer to both together). These additional datasets are still from an introductory programming course, but with a bigger range of exercises (in number and variety) with more students; a total of 208 students (spanning 10 distinct exercises) were contained in CS1-intro-10ex, whilst there were 313 students (spanning 280 distinct exercises) in CS1-var-2010, and 407 students (spanning 387 distinct exercises) in CS1-var-2011. Only six of the exercises were common between the two CS1-var datasets.

Much like the original CS1-intro-10ex dataset, the exercises in CS1-var are very straightforward (they are self-explanatory from their naming, e.g. 'maxNumber', 'isLeapYear', 'removeSpaces', 'getInitials', 'isPrime', 'addNumbers'). Finally, whilst the CS1-intro-10ex dataset contains 1593 submissions, the CS1-var-2010 and CS1-var-2011 datasets contain 3975 and 13849 submissions respectively. Due to the huge number of submissions in the CS1-var datasets, we manually validated only a random sample of 20% of the total submissions.

Table 1 shows the number of students who exhibit each indicator in the analysed CS1 datasets. IFRC is the most common in all the datasets, with 47%, 41% and 54% of students exhibiting this indicator in each respective dataset. For the CS1-var datasets, the IFASNV and IFASNB indicators follow next, at 22% and 21% respectively for CS1-var-2010, and 23% and 20% respectively for CS1-var-2011. For

Table 1: Number of students per indicator in each course

	CS1-intro-10ex	CS1-var-2010	CS1-var-2011	Design
Students (N)	208	313	407	43
UIF	29	13	30	1
IFRC	97	128	218	10
IFRT	73	21	39	12
IFASNV	30	69	95	4
IFASNB	7	63	80	0
EIFB	3	1	1	1
EEB	7	2	3	2
UNIF	48	5	16	11
CE	7	9	19	11
UE	5	10	11	1
UDIE	30	22	30	6
IUDIE	12	3	5	3
UDAD	5	32	13	2
UC	10	4	43	0
SV	9	1	6	0
LV	15	6	5	31

the CS-intro-10ex dataset, following IFRC are the IFRT and UNIF indicators with 35% and 23% respectively.

Table 2 shows the four most common semantic style indicators, and the percentage of students who submitted code that included these indicators for each of the 10 different exercises that formed the CS1-intro-10ex dataset. The exercise, "productIsEven", shows 37% of students exhibited the IFRC indicator. A large proportion of students also exhibited the IFRT indicator in the same "productIsEven" exercise, whilst UNIF is found mostly in "magicNumbers". The IFASNV indicator was largely limited to the "containersNeeded", "weeklyPay", "swapEnds", and "replaceCharAtPos" exercises.

Table 3 shows a different perspective, with the number of indicators per student expressed as cumulative frequency. The figures show how many students are "affected" by at least a certain number of indicators (and are therefore likely to benefit from some sort of feedback). For example, 90% of the students from CS1-intro-10ex exhibited at least one indicator, and 54% of the students exhibited at least two indicators. In the CS1-var datasets, 70% and 76% of the students exhibited at least one indicator (2010 and 2011 respectively).

After analysing over 19000 submissions from a total of 928 CS1 students, we decided to continue our investigation by examining a subset of projects of a design course. In particular, we have analysed an assignment of a software design course held in the fourth year of a highly competitive Bachelor in Software Engineering (that we called "Design") held at the University of Auckland. Students had to develop a small Java project with multiple packages and classes, where they developed the Kalah game [46]. Unlike the CS1 datasets, where students submitted their code using CodeWrite [18], these students could use their favourite IDE, and their submissions consisted of multiple Java source files. Of particular interest is the fact that the students expected their submission to be assessed according to "good design" (specifically, "modifiability") and so we might expect that they would pay attention to programming style as well. The Design dataset is composed of 43 submissions, one for each student. We manually validated this complete dataset.

Table 2: Percentage of top 4 semantic style indicators per exercise – CS1-intro-10ex

Exercise	**Stud**	**IFRC**	**IFRT**	**UNIF**	**IFASNV**
replaceCharAtPos	182	0%	0%	0%	1%
productIsEven	205	37%	30%	0%	0%
swapEnds	175	0%	0%	0%	4%
containersNeeded	182	0%	0%	0%	11%
weeklyPay	177	0%	0%	1%	6%
countOdds	138	0%	0%	0%	0%
posOfValInArray	132	0%	0%	1%	0%
reverse	130	0%	0%	0%	0%
magicNumbers	133	32%	15%	10%	0%
sumValues	130	0%	0%	30%	0%

Table 3: Cumulative frequency of number of indicators per student – CS1-intro-10ex, CS1-var-2010, CS1-var-2011, and Design have respectively 208, 313, 407, and 43 students in total

#Indicators	CS1-10ex	CS1-var-2010	CS1-var-2011	Design
7	0%	0%	0%	0%
6	0%	1%	1%	0%
5	2%	2%	2%	2%
4	8%	4%	8%	7%
3	21%	15%	19%	16%
2	54%	31%	42%	40%
1	90%	70%	76%	84%

We have only examined students of the University of Auckland, even though we have considered different courses and years. This raises the question as to whether what we have seen could be generalised to other student cohorts. There is nothing in our student groups that would suggest they are particularly prone to such semantic style issues. Moreover, the design course (Design) consisted in a number of international students along with students from the aforementioned highly competitive undergraduate programme. More than 50% of the students were in their fourth year of studies, had attended at least 10 courses involving programming, and so they were expected to be familiar with at least three programming languages: C, Python and Java.

As shown in Table 3, 84% of the fourth-year students exhibited at least one of these indicators, with 40% of these students exhibiting multiple indicators. This suggests that the indicators we have identified are useful for providing feedback to more than just complete novice programmers. It also suggests that the results we observed for the other datasets were not a consequence of the limitations place on how the students created their submissions (a simple web-browser application with little IDE support).

6 DISCUSSION

We have identified 16 *semantic style indicators* — statement-level indications of a potential lack of knowledge by a programmer. As Table 1 suggests, many of these indicators were found in all the datasets – even when dealing with fourth-year students. Also, a high proportion of students exhibited multiple indicators, suggesting that these indicators potentially represent lack of knowledge of one or more aspects of programming. It is worth noting that such lack of knowledge may not be only related to novice programmers, but to anyone inexperienced in a particular language. From this, we can claim an affirmative answer to RQ1.

We believe the importance of semantic style indicators is that their presence may indicate a source of confusion regarding some programming concepts. If this is true, then being able to detect them can lead to the provision of useful feedback. That they can be quickly detected automatically means this feedback will be able to be provided early, and without the need for direct intervention by teachers.

We do not yet have a direct link between the presence of indicators and lack of understanding. We will investigate this in future work. It could be that there is no link. For example, what we see is due to expediency on the part of the student, rather than lack of understanding. Or it could be that this is just a consequence of students still coming to grips with some concepts, and they quickly grow out of using such indicators.

However, indicators generally require more work to use rather than more reasonable alternatives, which argues against their use being due to laziness or lack of time. The number of occurrences of the indicators, especially the number of students who exhibit them, also argues against expediency. The fact that even fourth-year students exhibit the indicators argues against indicators being just a phase students go through. So our results do provide indirect evidence that semantic style indicators suggest there is some issue that we as teachers must address.

Our analysis was done on the AST representation of the student submissions, that is, it did not require any expensive processing. In fact analysing the biggest dataset (almost 14000 files) took less than 400 seconds on an rMBP L13 i7 16 GB RAM (and it was probably mostly due to file operations). From this, we can claim that detecting the semantic style indicators we have identified is not expensive from a computational perspective (RQ2).

As we noted in section 4, two indicators we identified (Selfish Variable and Lonely Variable) are already reported by many IDEs. That our methodology identified issues regarded as problematic by the community lends weight to our claim that the other indicators we have identified should be made known to programmers. We suggest there is value in distinguishing between syntactic and semantic issues in such tools, especially in a teaching environment.

Some indicators (e.g. IFRC) occur very frequently across all the datasets. Others (e.g. EIFB, EEB) are not as frequent, but this does not mean that they are less useful. Rather, broadening the types of indicators is likely to prove helpful in providing useful feedback not only to a larger set of students, but also to programmers who may be unfamiliar with a particular language/construct. Table 3 indicates the potential impact of this approach by identifying the percentage of students who exhibit at least "n" indicators. In particular, in the CS1-intro-10ex dataset 90% of the students exhibit at least one of the 16 semantic style indicators, while 54% of the students exhibit two or more indicators. The CS1-var datasets in Table 3 show similar impact potential, with respectively 70% and 76% of students showing at least one indicator, while 31% and 42% of students show at least two indicators. In the Design dataset, 84% of students exhibted at least one indicator, while some of these experienced students showed up to five indicators. The latter is an important point, because they are fourth-year students and our dataset is very small (only 43 students). This suggests there is value to providing feedback on semantic issues even for experienced programmers.

Finding more semantic style indicators, and organising them in a useful manner [13], will be the subject of future work. We also plan to integrate indicator detection and feedback into tools used by students. Feedback could be a simple suggestion on how the code can be refactored (and why) without enforcing any further step. The challenge is to provide the feedback, as soon as possible as advocated by Murphy-Hill and Black [34], to be a source of information to instructors with the aim of helping to shape teaching [35], and to reduce the formation of long-term habits as noted by Ala-Mutka et al. [10].

For those indicators for which there were a small number of occurrences, it may be that it is the same student creating them, which would exaggerate their importance. We manually examined these cases and confirm that this was not the case. For example, as shown in Table 1, seven instances of the EEB indicator were found in the work of seven distinct students of CS1-intro-10ex, two instances of the same indicator were found in the work of two distinct students of CS1-var-2010, and three instances of the same indicator were found in the work of three students of CS1-var-2011.

Futhermore, we manually validated the datasets where possible, i.e. we carefully looked at the source code of 20% of the results to confirm the presence of the indicator(s) detected. Due to the large number of submissions we validated only those indicators

identified by the tool (true positives and false positives). We have not yet validated whether there are submissions undetected by our tool that contain one or more of the 16 indicators (false negatives). The presence of such would further underline the importance of providing feedback for these issues.

We have only examined students of our university, even though we have considered different courses and years. This raises the question as to whether what we have seen could be generalised to other student cohorts. There is nothing in our student groups that would suggest they are particularly prone to such semantic style issues. Moreover, the design course (Design) consisted in a number of international students along with students from the undergraduate programme at our university. More than 50% of the students were in their 4th year of a Bachelor in Software Engineering.

Finally, our datasets contain only Java code. While there is nothing language-specific in our choice of language indicators, they may present differently in other languages or not apply (e.g. because the language does not have a `Boolean` type).

7 CONCLUSION AND FUTURE WORK

We have proposed the concept of *semantic style indicators*, which are statement-level characteristics that may indicate a lack of knowledge of some aspect of programming. We have identified 16 semantic style indicators from manual analysis of student solutions to different exercises. We developed a tool to detect these indicators by analysing their AST, and applied the tool to more than 19000 submissions from 928 undergraduate students who solved more than 665 distinct exercises. It was not only found that a high number of the submissions contained these indicators, but also that most students at all levels exhibited one or more of these indicators. This demonstrates that not only are semantic style indicators found in students, but they are in fact quite common and even the simplest of programs could contain more than one.

We believe the importance of semantic style indicators is that their presence may indicate a source of confusion regarding some programming concepts. At the very least, our results suggest that there is something going on that we as teachers must understand in order to improve how we teach programming.

Semantic style indicators represent a first step towards providing useful formative feedback on programming style. Once an indicator is detected, feedback can be provided on where the indicator is, what the potential issue is, and how the code could be improved. Finding semantic style indicators requires little computation power, thus our method can easily be used or integrated into IDEs. Future work will focus on defining a catalogue of indicators, finding more of them, and applying our analysis to an even larger corpus of code and number of students. The scope and definition of the indicators might be further expanded when looking at more complex exercises. Our current work in progress includes an IDE plugin for detecting indicators and providing feedback. We plan to evaluate this plugin with programmers (including those in industry) to determine its usefulness, and demonstrate the efficacy of the "semantic style indicator" concept in general.

REFERENCES

[1] 2017. Checkstyle. http://checkstyle.sourceforge.net. (2017). [Online; accessed September 2017].

[2] 2017. Error Prone. http://errorprone.info/. (2017). [Online; accessed September 2017].

[3] 2017. Findbugs. http://findbugs.sourceforge.net. (2017). [Online; accessed September 2017].

[4] 2017. Infer. http://fbinfer.com/. (2017). [Online; accessed September 2017].

[5] 2017. PMD. https://pmd.github.io. (2017). [Online; accessed September 2017].

[6] 2017. Sonar. http://www.sonarqube.org. (2017). [Online; accessed September 2017].

[7] ACM and IEEE. 2013. *Computer Science curricula 2013: Curriculum guidelines for undergraduate degree programs in Computer Science.*

[8] ACM and IEEE. 2014. *Software Engineering 2014: Curriculum guidelines for undergraduate degree programs in Software Engineering.*

[9] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. 2015. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the 11th Annual International Conference on International Computing Education Research (ICER)*. ACM, 121–130.

[10] Kirsti Ala-Mutka. 2005. A Survey of automated assessment approaches for programming assignments. *Computer Science Education* 15, 2 (2005), 83–102.

[11] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. 2004. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education* 3, 1 (2004), 245–262.

[12] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering* 21, 1 (2016), 104–158.

[13] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Vol. 1. IEEE, 470–481.

[14] Neil Christopher Charles Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the 10th annual conference on International computing education research (ICER)*. ACM, 43–50.

[15] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education (SIGCSE)*. ACM, 223–228.

[16] Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, and Andy Zaidman. 2017. UAV: Warnings from multiple automated static analysis tools at a glance. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 472–476.

[17] Andrew Cain and Muhammad Ali Babar. 2016. Reflections on applying constructive alignment with formative feedback for teaching introductory programming and software architecture. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE-C)*. ACM, 336–345.

[18] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. CodeWrite: supporting student-driven practice of Java. In *Proceedings of the 42nd ACM technical symposium on Computer science education (SIGCSE)*. ACM, 471–476.

[19] Eclipse. 2017. Java Development Tools (JDT). http://www.eclipse.org/jdt/. (2017). [Online; accessed September 2017].

[20] Stephen Edwards, Nischel Kandru, and Mukund Rajagopal. 2017. Investigating static analysis errors in student Java programs. In *Proceedings of the 13th ACM Conference on International Computing Education Research (ICER)*. ACM, 65–73.

[21] Martin Fowler. 2017. Code Smell. http://martinfowler.com/bliki/CodeSmell.html. (2017). [Online; accessed September 2017].

[22] M. Fowler and K. Beck. 1999. Refactoring: improving the design of existing code. *Addison-Wesley Professional* (1999).

[23] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. 2014. Some code smells have a significant but small effect on faults. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (Sept. 2014), 33:1–33:39.

[24] Colin Higgins, Pavlos Symeonidis, and Athanasios Tsintsifas. 2002. The marking system for CourseMaster. *ACM SIGCSE Bulletin* 34, 3 (2002), 46–50.

[25] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, Foutse Khomh, and Mohammad Zulkernine. 2016. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering* 21, 3 (2016), 896–931.

[26] David Jackson and Michelle Usher. 1997. Grading student programs using ASSYST. In *ACM SIGCSE Bulletin*, Vol. 29. ACM, 335–339.

[27] Sebastien Jeanmart, Yann-Gael Gueheneuc, Houari Sahraoui, and Naji Habra. 2009. Impact of the visitor pattern on program comprehension and maintenance. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 69–78.

[28] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 110–115.

[29] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a systematic review of automated feedback generation for programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer*

Science Education (ITiCSE). ACM, 41–46.

[30] Theodora Koulouri, Stanislao Lauria, and Robert Macredie. 2015. Teaching introductory programming: a quantitative evaluation of different approaches. *ACM Transactions on Computing Education (TOCE)* 14, 4 (2015), 26.

[31] Xiaosong Li and Christine Prasad. 2005. Effectively teaching coding standards in programming. In *Proceedings of the 6th conference on Information technology education*. ACM, 239–244.

[32] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 192–201.

[33] Naouel Moha, Yan-Gael Gueheneuc, Laurence Duchien, and Anne Francoise Le Meur. 2010. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (Jan. 2010), 20–36.

[34] Emerson Murphy-Hill and Andrew P. Black. 2008. Seven habits of a highly effective smell detector. In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*. 36–40.

[35] David Nicol and Debra Macfarlane-Dick. 2006. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in higher education* 31, 2 (2006), 199–218.

[36] Claudia Ott, Anthony Robins, and Kerry Shephard. 2016. Translating principles of effective feedback for students into the CS1 Context. *ACM Transactions on Computing Education (TOCE)* 16, 1 (2016), 1.

[37] Fabio Palomba, Andrea De Lucia, Gabriele Bavota, and Rocco Oliveto. 2015. Anti-pattern detection: methods, challenges, and open issues. *Advances in Computers* 95 (2015), 201–238.

[38] Kelly Rivers, Erik Harpstead, and Ken Koedinger. 2016. Learning curve analysis for programming: Which concepts do students struggle with?. In *Proceedings of the 12th ACM Conference on International Computing Education Research (ICER)*. ACM, 143–151.

[39] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education* 13, 2 (2003), 137–172.

[40] Tom Schorsch. 1995. CAP: an automated self-assessment tool to check pascal programs for syntax, logic and style errors. In *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*. ACM, New York, NY, USA, 168–172.

[41] Dag Sjøberg, Aiko Yamashita, Bente Cecilie Dahlum Anda, Audris Mockus, and Tore Dybå. 2013. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering* 39, 8 (Aug. 2013), 1144–1156. https://doi.org/10.1109/TSE.2012.89

[42] Herb Sutter and Andrei Alexandrescu. 2004. *C++ coding standards: 101 rules, guidelines, and best practices*. Pearson Education.

[43] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and removal of Type-Checking bad smells. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*. 329–331.

[44] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your Code starts to smell bad. In *the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. 403–414.

[45] Grant Wiggins. 2012. Seven keys to effective feedback. *Feedback for Learning* 70, 1 (2012), 10–16.

[46] Wikia. 2017. Kalah Game. http://mancala.wikia.com/wiki/Kalah. (2017). [Online; accessed September 2017].

[47] Laurie Williams, Robert Kessler, Ward Cunningham, and Ron Jeffries. 2000. Strengthening the case for pair programming. *IEEE software* 17, 4 (2000), 19.

[48] Aiko Yamashita and Leon Moonen. 2012. Do code smells reflect important maintainability aspects?. In *the 28th IEEE International Conference on Software Maintenance (ICSM)*. 306–315. https://doi.org/10.1109/ICSM.2012.6405287