


Soploon: A virtual assistant to help teachers to detect object-oriented errors in students' source codes

Sebastián Vallejos | Luis S. Berdun  | Marcelo G. Armentano | Álvaro Soria | Alfredo R. Teyseyre

ISISTAN Research Institute, (CONICET-UNICEN), Campus Universitario, Paraje Arroyo Seco, Tandil, Buenos Aires, Argentina

Correspondence

Luis S. Berdun, ISISTAN Research Institute, (CONICET-UNICEN), Campus Universitario, Paraje Arroyo Seco, Tandil 7000, Buenos Aires, Argentina.
Email: luis.berdun@isistan.unicen.edu.ar

Abstract

When checking students' source codes, teachers tend to overlook some errors. This work introduces Soploon, a tool that automatically detects novice programmer errors. By using this tool, teachers can reduce the number of overlooked errors. Thus, students receive a more complete and exhaustive feedback about their errors and misconceptions.

KEYWORDS

declarative meta-programming, detect object-oriented errors, novice programmer errors, students' misconceptions, teaching object-oriented programming

1 | INTRODUCTION

It is well known that teaching Object Oriented Programming (OOP) is a challenging task [32]. In fact, some authors consider the task of teaching programming abilities more difficult than the task of teaching physics, calculus, or chemistry [16]. The process of learning programming usually presents a wide range of difficulties to students and novice programmers [30], who have to understand not only program control structures, such as loops or if-then-else statements, but also complex concepts, such as backtracking and recursion. Moreover, Object Oriented Programming adds extra complexity to students, by introducing concepts such as classes, objects, inheritance and polymorphism, among others [5,25].

Teaching programming with activities that only involve reading or listening is not an effective way to teach students how to program correctly. This kind of activities does not stimulate higher-order thinking, since students remain as passive recipients of information. Therefore, teaching OOP by the exclusive use of traditional reading lessons may hinder the learning process of students [2]. In OOP, students not only have to understand programming concepts, but also have to

learn how apply them in an effective way [8,26,32]. In this sense, students need to practice and do programming exercises to learn how to program correctly [8,32]. Using and doing activities, like programming, have a significant positive impact on learning and stimulate high-order thinking [3]. In fact, a survey involving six universities [15] revealed that both, teachers and students, consider learning programming by doing more effective than by reading or listening.

Consequently, during programming courses, students usually must complete programming tasks and a coursework. By doing programming exercises, students can apply the concepts that they learned in the course. Additionally, since students tend to overestimate their understanding [15], assessing coursework allows teachers to detect students' misconceptions that would be impossible to detect in lecture sessions. Identifying and addressing students' misconceptions is crucial to effective teaching [22,28]. In this way, teachers can reinforce in future lessons those programming concepts that the students failed to understand. The main disadvantage of the coursework method is that teachers individually have to assess each of the students' source code. If students are learning programming control structures,

assessing coursework can be semi-automated by running students' code through a set of test cases. However, if students are learning the object-oriented programming paradigm, test cases are useless: a student program can give the expected outputs for every test case, but still may have several object-oriented programming errors. In this case, teachers have to manually check and assess students' source codes.

Manually assessing student's source codes is a time consuming task. When assessing a student' source code, teachers have to check both the whole design of the program and each individual line of code wrote by the student. Assessing only one source code may be a simple task for an experienced programming teacher, but when a teacher has to asses dozens of works, it may become a tedious and stressful task. At this point, teachers may start to not paying the enough attention and overlook some student errors. For example, declaring a *public* field is a serious mistake in object-oriented programming, but it can easily be overlooked since it is just one line of code among several classes and hundreds of lines of code.

An incomplete feedback of the students' source codes may hinder the students learning process. If the teacher does not detect and address the student's misconceptions on time, the student will learn wrong concepts taking them as correct. When an student learns a new concept, he/she builds a mental model to give meaning to that concept [6]. In this sense, if the student builds a wrong mental model about OOP concepts, it will be necessary to rebuild that model with the correct concepts. The main problem is that rebuilding mental models (unlearn and relearn) is not an easy task for human beings [1]. In fact, previous experiments revealed that most students are reluctant to dismiss their wrong mental models [10]. To avoid this problem, it is crucial to identify and address the students' misconceptions before students build wrong mental models. For the reasons described above, every time the teacher assess a student's source code (either of the coursework or a programming task), the assessment must be exhaustive and the feedback given to students must be as complete and detailed as possible.

In this work, we address this problem by introducing Soploon, an assistant that helps teachers in the students' codes assessment task. This assistant automatically detects object-oriented errors in the students' codes and notifies teachers about them. Moreover, the assistant is customizable, so the teacher can specify the particular errors that his/her students are used to making in order to detect them. After detecting an error, the assistant is capable of locating and highlighting the code fragment containing the error. Thus, the teacher can easily find the students' errors to give each student a complete and appropriate feedback about his/her source code. In order to provide its functionality, Soploon uses declarative meta-programming (DMP). DMP consists in the use of a declarative programming language at a meta-level to reason

about and manipulate programs built in some underlying base language [4]. In particular, the assistant uses this technique to reason about students' source codes and detect the errors defined by the teacher.

The rest of the article is organized as follows. Section 2 describes the related works. Section 3 introduces how Soploon can assist teachers in the task of detecting errors in students' source codes. Section 4 presents an overview of the main design components of Soploon. Section 5 discusses some experiences and experiments carried out to test the assistant with teachers. Finally, section 6 outlines the conclusions of the article as well as some lines about future work.

2 | RELATED WORK

The automatic detection of errors in a program source code has been previously addressed by several approaches. Some of these approaches calculate software metrics looking for atypical values that indicate the presence of a bad smell. For example, ref. [29] proposed the use of a distance metric based on the cohesion between artifacts in order to detect and refactor three bad smells: feature envy, low cohesion and lazy class. In a similar way, [24] calculates the degree of coupling between the artifacts of the project to detect two bad smells: shotgun surgery and divergent change. Other works [13,14,21] have used source code metrics (such as the number of methods per class and the number of lines of code per method, among others) in order to detect different bad smells. In ref. [17], authors presented iPlasma, a tool that materializes the use of source code metrics for the detection of bad smells. However, detecting bad smells by using software metrics is subject to interpretation and it is usually imprecise [20,31]. Moreover, metrics-based tools are not able to detect some errors that depend on the program structure or semantic (such as a public field that is breaking the object encapsulation) [12,20].

Other approaches address the detection of bad smells by using regular expressions or heuristics. In ref. [11] authors presented Espresso, a pre-compiler tool that tokenizes the source code and looks for novice errors that prevents the code for compiling. However, this tool focuses on enriching the error messages generated by the compiler, and not on detecting object oriented errors. Jsmell [27] is a tool for detecting bad smells in Java code. JSmell is able to detect seven different bad smells using a specific heuristic for each one. In a similar way, there are more tools (such as Jdeodorant¹ and PMD²) that use heuristics for detecting different bad smells. However, these tools may provide different results when they analyze the same system, usually because they have different interpretations for a same bad smell [9]. Moreover, these tools do not allow to modify the

interpretation of a specific bad smell or to add new errors to be detected. To the best of our knowledge, only PMD allows adding new errors to be detected, but it is a tedious task intended for engineers familiar with Java or XPath, which limits its use to a wider audience [19].

Finally, some approaches proposed the use of declarative meta-programming (DMP) to detect bad smells and design flaws. These approaches translate the source code to be analyzed into a representation in a meta-level language, and then reason about this representation to detect bad smells. In ref. [23] authors used DMP to detect refactoring opportunities for four bad smells. However, during the creation of the representation in the meta-level language, this work only considers high level aspects (such as classes, fields, etc.) and omits most low level statements (such as loops and if conditions, assignments, method variables, etc.). Therefore, this approach cannot detect some errors, such as the use of *instanceOf* in an *if-statement*. Other works refs. [18,31] have proposed the creation of a more detailed meta-level representation. Nevertheless, these works focus on the detection of a small set of bad smells that negatively affect the maintainability of commercial software, such as obsolete parameters and inappropriate interfaces [31], message chains, inappropriate intimacy, and middleman [18]. These errors do not include common errors that novice programmers and students are used to making. For example, using *super* to access a field already inherited from the parent class is an error that does not significantly affect software maintainability, but it reflects that the student did not understand the concept of inheritance. In ref. [12], the authors suggested that the use of DMP to detect bad smells would be useful in software development training.

In contrast to previous works that only detect a small set of bad smells that negatively affect the maintainability, our approach currently detects an initial set of 47 frequent errors that novice programmers and students generally make. Moreover, most commercial tools for detecting bad smells only detects a fixed number of bad smells whereas our

approach is customizable: the teacher can easily specify new errors to be detected, and he/she also can modify or delete any of the 47 errors already specified in the tool. Thus, the teacher is able to personalize the tool's interpretation about what is an error and what is not. Finally, the tool can highlight the code fragments containing each detected error, this is a useful feature for teachers that some other tools lack [9].

3 | SOPLOON

Soploon is intended for assisting teachers in assessing the student's source codes in order to provide a more exhaustive feedback about the students misconception regarding object oriented programming. The assistant uses Prolog as a meta-level language to analyze the Java source codes. When the assistant detects an error, it notifies the teacher and highlights the code fragment that contains the error. In this way, the teacher can easily find errors in the student's source code and give a more complete feedback to the student. Moreover, it is possible for the teacher to specify new errors to be detected by the assistant. In this way, the teacher can customize the assistant to detect the particular errors that his/her students are used to making.

Figure 1 shows an overview of the assistant's workflow. On the one hand, the assistant translates the student's source code into a representation in Prolog (a). During this process, each syntactic construct of the Java source code is represented as a Prolog fact. Once the Prolog facts are generated, the assistant detects errors in the student's source code by logic inference (b). To do this, the assistant executes queries over the Prolog facts generated during step (a). For each query, the assistant tries to find a set of Prolog facts that satisfies the conditions declared by a Prolog rule that specifies the preconditions for a particular error. The tool has an initial set of 47 common OOP error specified as Prolog Rules. Additionally, the teacher can specify new OOP errors to be

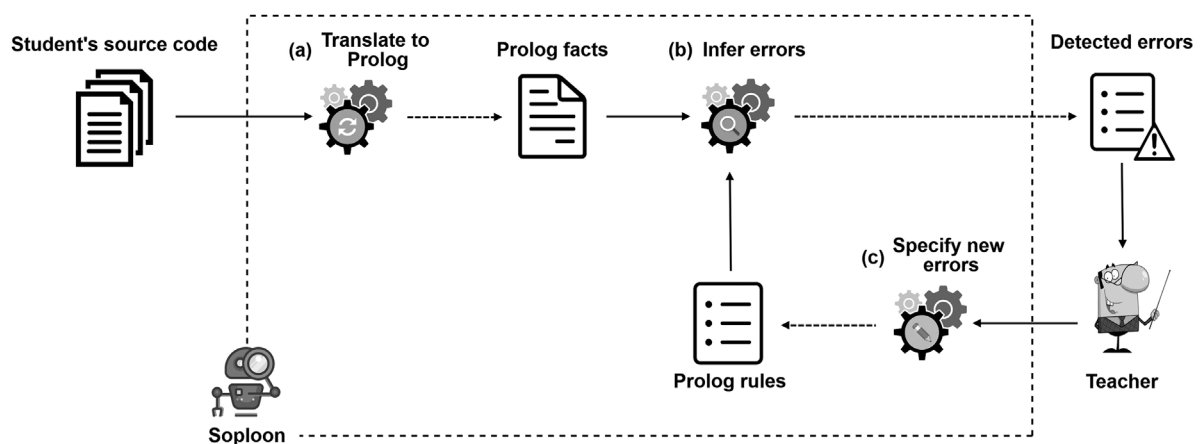


FIGURE 1 Conceptual model of Soploon

```

1
2 public class MyClass {
3
4     public int my_field;
5
6 }
7

```

FIGURE 2 Screenshot of the source code example in Eclipse

detected as Prolog Rules (c), enabling the teacher to add domain rules for a particular problem besides OOP general errors. For example, the teacher can add a rule stating that having a variable with the name “type” is considered an error. If a set of Prolog facts satisfies the conditions of a Prolog rule, then those Prolog facts contains the error described by that rule. When an error is detected, the assistant notifies the teacher and highlights the corresponding Java code fragment containing the error.

Soploon was materialized as an Eclipse³ plugin. In this way, the teacher can open the student's source code as an Eclipse project, and ask the assistant to detect errors in it. It is important to note that there were no particular reasons for selecting Eclipse as IDE, Prolog as meta-level language, or Java as base language, more than our familiarity with these technologies and their popularity. However, the assistant could be developed to work with any other languages and over any other IDE (even as a standalone assistant).

The next sub-sections detail how the assistant carries out the three steps described in Figure 1. In order to clearly explain these steps, Figure 2 introduces a Java source code with a common error for novice students. The code declares a class named “MyClass” with a public field named “my_field.” Declaring public fields breaks encapsulation, which is considered to be a serious error in object-oriented programming. The following sub-sections use this simple Java code to exemplify each step of the assistant's workflow.

3.1 | Translate to Prolog (step A)

In this step, the assistant translates the student's source code to Prolog facts. The first part of this step consists in generating the abstract syntax tree (AST) of the student's code. An AST is a tree representation of the syntactic structure of a source code. Each node of this tree represents a syntactic construct of the source code. Analyzing the student's source code by walking its AST is simpler than analyzing the plain source code itself. For this reason, when the teacher asks for detecting errors on a student source code, the assistant first creates the abstract syntax tree for that code.

Figure 3 shows the AST generated from the Java code presented in Figure 2. On the left is the student source code and its structure. On the right is the AST generated for that code. The root of the AST is node “a,” which represents the compilation unit (i.e., the Java file). This node has node “b” as its only child, representing the class declaration of “MyClass.” Its child named node “d” represents the field declaration statement within “MyClass.” A single field

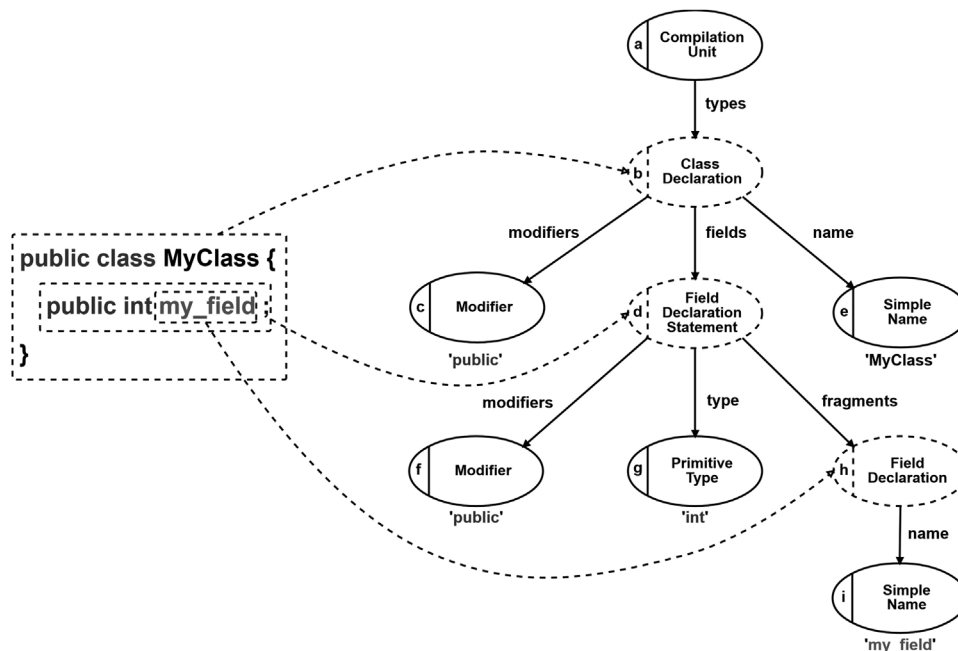


FIGURE 3 Abstract syntax tree for the sample code

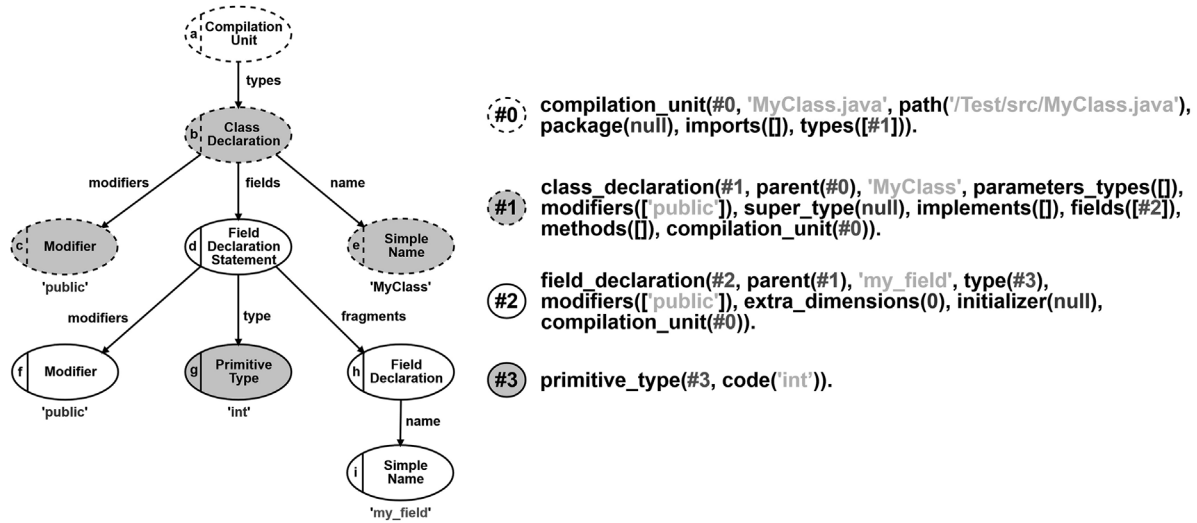


FIGURE 4 Prolog representation of the sample code

declaration statement can define multiple fields. In this particular case, it defines only one field: “my_field.” This field is represented by node “h.” The remaining nodes “c,” “e,” “f,” “g,” and “i” represent simple data (such as names and modifiers) and they have no children nodes.

Once the AST is created, the second part of the translating step consists in walking down the tree visiting each node and generating the corresponding Prolog facts. From each node the assistant extracts the kind of syntactic construct that the node represents and its property values. The information extracted from a single node or from a set of nodes is used to create prolog facts corresponding to a meta-level representation of the student's source code.

Figure 4 shows the Prolog facts generated from AST presented in Figure 3. On the left is the AST of the student's source code. On the right is the set of Prolog facts generated from the AST. The border line and the fill color of each AST node indicates which Prolog fact contains the information of that node. For example, the fact #0 contains the information extracted from the node “a.” We can observe that some facts contain information from multiple related nodes. For example, fact #1 contains information extracted from nodes “b,” “c,” and “e.” In this way, the final number of facts is reduced, improving the performance during the error detection step.

Each Prolog fact is composed of three parts: a predicate name, an identifier, and a set of arguments. The predicate name indicates the kind of structure that the fact represents. For example, fact #1 represents a class declaration. The first argument of the fact corresponds to the identifier. It allows creating references between different facts in order to respect the hierarchical structure of the AST. Continuing with the example, fact #1 has id “#1”, and its parent is fact #0 (i.e., the compilation unit). Finally, the rest of the arguments contain the information extracted from the AST nodes. The number of

arguments depends on the kind of structure represented by the fact.

At the end of this step, the assistant has a Prolog representation of the original source code. Although the presented example is very simple, it shows the translation process from Java to Prolog. The reason for using a simple example is that as the source code grows, the AST becomes more complex and the final number of Prolog facts increases. Actually, it was necessary to define 74 different predicate names⁴ to represent each possible Java syntactic construct as a Prolog fact. The assistant translates to Prolog not only the high level aspects of the source code, such as type declarations, but also low level statements, such as method invocations, loops, logic conditions, and arithmetic operations, among others. In this way, the assistant can accurately represent the whole structure and semantic of the original Java source.

3.2 | Infer errors (step B)

In this step, the assistant detects object oriented errors in the student's source code. To do this, the assistant has a predefined set of Prolog rules. Each rule defines the needed conditions for a specific object-oriented error to occur. Continuing with the public field example, Figure 5 shows the Prolog rule for the detection of the “public field” error. In the

```
public_field(X):-
    ① field_declaration(X, _, _, modifiers(M), _, _, _, _),
    ② contains(M, 'public'),
    ③ not(contains(M, 'final')),
    ④ not(contains(M, 'static')).
```

FIGURE 5 Prolog rule for detecting a public field

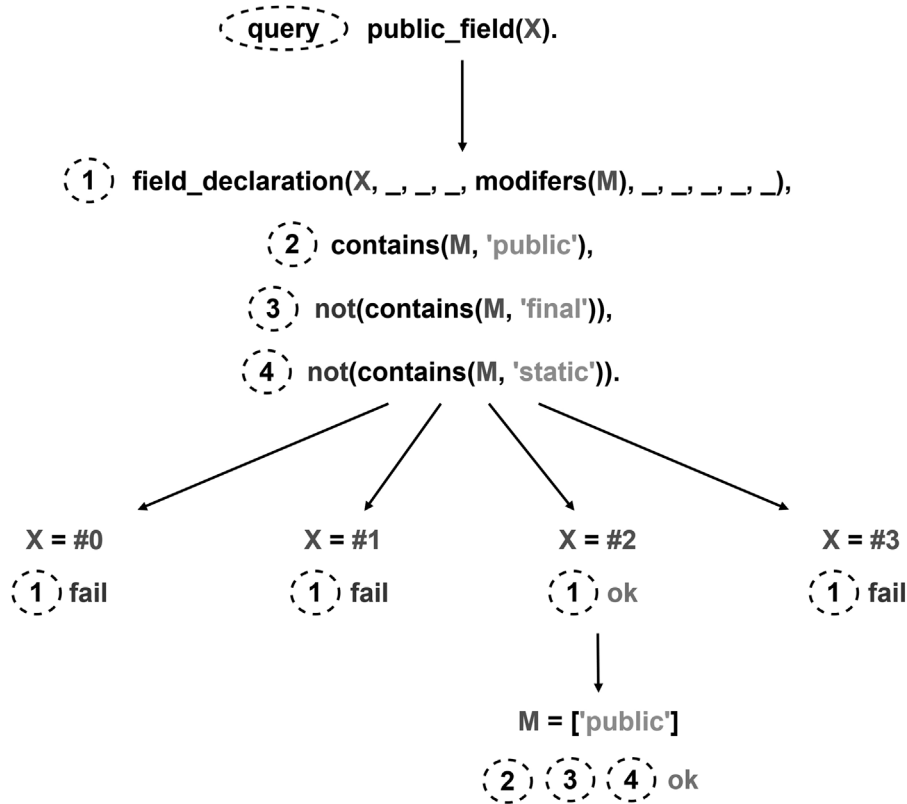


FIGURE 6 Execution trace of the Prolog rule for detecting a public field in the automatic generated prolog facts

rule, X is a variable that represents the identifier of a Prolog fact to be evaluated (such as #1, #2, etc.). The rule defines four conditions to detect the public field error in the evaluated Prolog fact: (1) it must represent a field declaration; (2) it must have the “public” modifier; (3) it must have not the ‘final’ modifier; and (4) it must not have the “static” modifier. If a Prolog fact meets with these four conditions, then it contains the “public field” error.

With this set of Prolog rules, the assistant detects errors in the students’ source code by executing logic queries over the Prolog facts generated in the previous step (Subchapter 3.1). Each query looks for a Prolog fact that meets the conditions declared by some Prolog rule of the predefined set. When a fact meets with the conditions of a rule, then that fact contains the error specified by that rule. Figure 6 shows how the assistant detects the error on the Prolog facts from Figure 4 by considering the Prolog rule

defined in Figure 5. The assistant executes the query “public_field(X)” trying to find a Prolog fact that meets the rule conditions. Figure 6 details the execution trace for that query. As the Figure shows, the query evaluates the predicate “public_field” for the four Prolog facts #0, #1, #2, and #3. The facts #0, #1, and #3 do not meet the condition (1) since they do not represent a field declaration. However, fact #2 meets the four conditions: (1) it represents a field declaration; (2) it has the “public” modifier; (3) it has not the “final” modifier; and finally, (4) it has not the “static” modifier. Then, the assistant detects the “public_field” error on Prolog fact #2.

Once the assistant detects errors in the Prolog facts, the second part of this step consists in identifying the code fragment that contains those errors. Continuing with the example, Figure 7 shows a screenshot of the assistant highlighting the public field error in the Java source code from

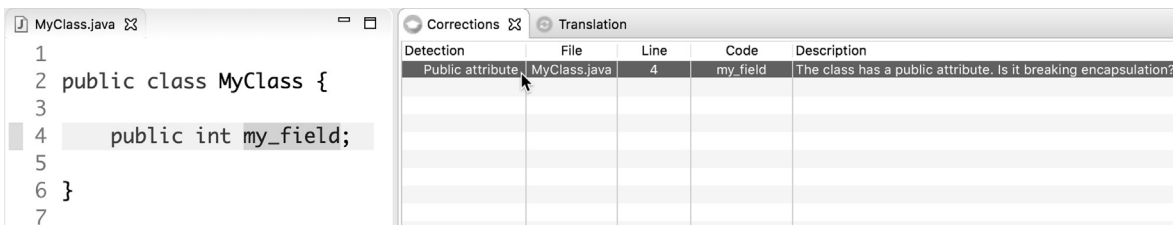


FIGURE 7 Screenshot of Soploun highlighting an error in the source code

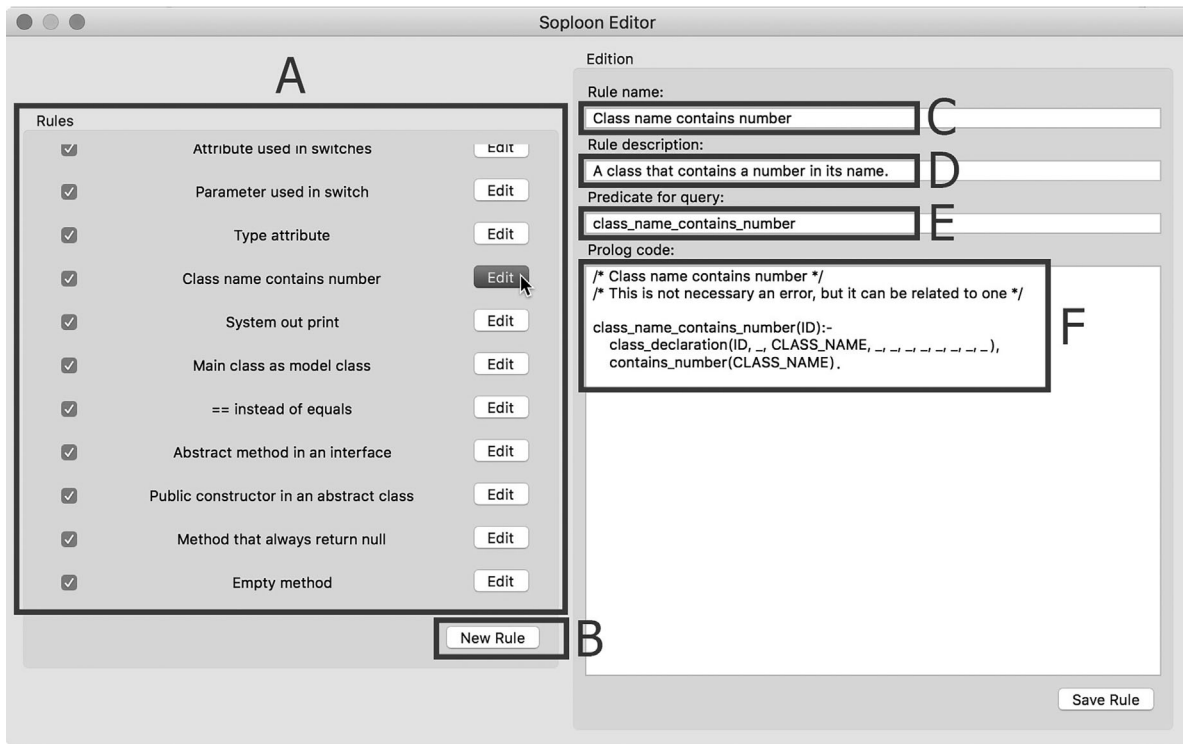


FIGURE 8 Screenshot of Soploon's customization window

Figure 2. After inferring errors, the assistant lists them in the “Corrections” tab. By double-clicking one of these errors, the assistant traces the selected error from the Prolog facts on which it was detected (Prolog fact #2 in the example) to the respective AST nodes. Then, it highlights the code fragment corresponding to those AST nodes (the field declaration in the example). Thus, the assistant helps the teacher to easily find errors in the student's source code.

3.3 | Specify new errors (step C)

In this step, the teacher can specify new object-oriented errors in addition to those already predefined in the assistant. Although, the assistant already has a set of 47 predefined errors, it is possible for the teacher to specify new errors or to modify the existing ones. In this way, he/she can customize the assistant to detect some particular errors that his/her students are used to making. To do this, the teacher has to define a Prolog rule with the conditions for the error to occur. Due to the great expressive power of Prolog, the teacher can specify any kind of programming error: from design errors (such as lazy classes or sister classes with duplicated code), to one-statement errors (such as public field declarations). Moreover, the teacher can specify not only object-oriented errors, but also program structures potentially related to student's misconceptions about object-oriented concepts.

As an example, Figure 8 shows a screenshot of Soploon's customization window where the teacher specifies a new rule

to detect classes containing numbers in their names. This is not necessarily an error, but it is usually related to student's difficulties to distinguish the concept of class from the concept of instance. For example, a student can define the classes “DiscountOf10” and “DiscountOf20,” instead of defining just the class “Discount” with a field to indicate the percentage of discount. The screenshot in Figure 8 shows on the left all the rules already defined (A) and gives the possibility to specify new rules (B). When specifying a new rule, the teacher has to define an identifying name (C), an optional brief description (D), a main predicate (this is, the name of the predicate to be used to infer errors), and the Prolog code needed for that rule (F). By defining the rule shown in Figure 8, the assistant will notify the teacher every time a class name contains numbers. Note that the Prolog rule presented in Figure 8 uses an auxiliary predicate named *contains_numer*. This predicate is part of a predefined set of generic auxiliary predicates⁵ included in Soploon to facilitate the teacher's work when specifying new rules. This set of predicates can be also customized to fit the teacher's needs.

4 | DESIGN OVERVIEW

Figure 9 shows an overview of the main components of Soploon and how they are connected to each other. There are two main packages: the user interface and the model. The first package groups the components related to user-interaction.

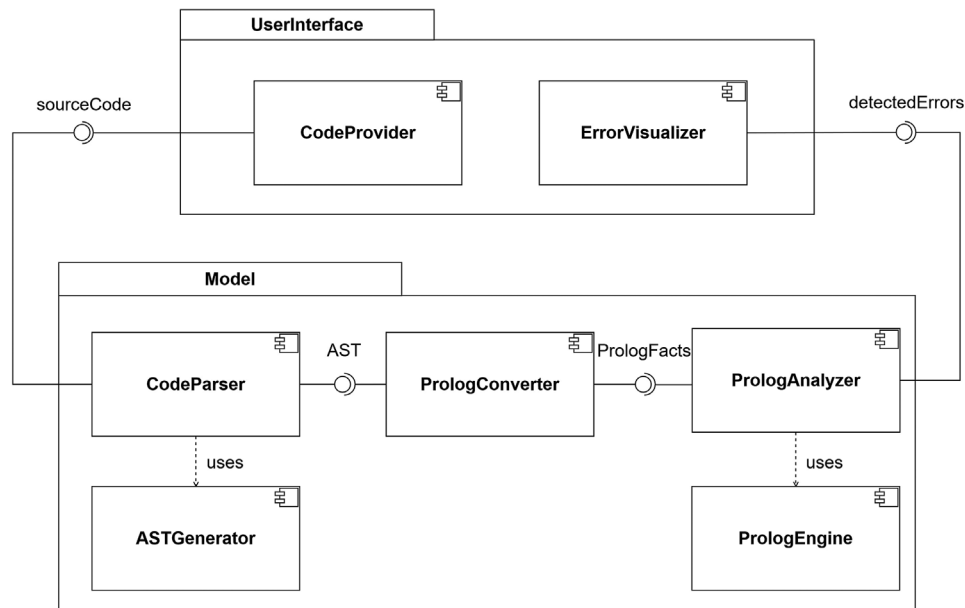


FIGURE 9 Design overview of Soploon

The second package groups the main components related to the code analysis. When the assistant is executed, the CodeProvider component is responsible for obtaining the source code to be analyzed. The CodeParser component parses the source code and generates the AST by using the ASTGenerator component. The PrologConverter component walks down the generated tree and creates the Prolog representation of the student's source code. Finally, the PrologAnalyzer component takes the Prolog representation and infers the errors specified by the teacher. The detected errors are shown to the user by the ErrorVisualizer component.

As we already mentioned, Soploon was materialized in Java as an Eclipse plugin. Using popular technologies facilitated the implementation of Soploon due to the existence of a large number of libraries and frameworks. The CodeProvider and ErrorVisualizer components were developed by using the Eclipse Java development tools⁶ (JDT). JDT provides the needed functionalities for extending Eclipse IDE such as adding components to the Eclipse user interface.⁷ The CodeParser and the ASTGenerator components were implemented by using the components provided by Eclipse framework.⁸ Finally, the PrologEngine component was materialized using tuProlog [7], which is a light-weight Prolog engine developed in Java.

5 | EXPERIMENT

An experiment was carried out to assess the behavior of Soploon in a real environment. The experiment consisted in using the assistant to check students' coursework that had been previously checked by teachers. The main objective of this experiment was to compare the corrections made by

experienced teachers with the corrections made by the assistant in order to response the following research questions: *Can the assistant detect errors overlooked by the teachers? Can the assistant help to reduce teachers' workload? Can the assistant replace the teacher in the corrections of the works?*

The experiment took place in an object-oriented programming course at UNICEN University (Argentina). In this course, teachers introduce object-oriented programming to students who already know basic programming structures (such as loops and if-statements). To promote the course, the students need to approve a coursework besides the classical exam. This coursework consists in a work statement that students have to solve by applying the object-oriented programming concepts that they have learned during the course. Before the exam, teachers check each student's work and provide feedback. This feedback is useful for students because it allows them to correct conceptual errors (i.e., the code compiles but it contains OOP misconceptions) before the exam. For this reason, the feedback should be as complete and exhaustive as possible.

The experiment consisted in two parts. First a set of 15 student's works were corrected by human teachers. To avoid bias in the evaluation we used the corrections made by two teachers that did not participated in the development of the teacher assistant. Thus, the way in that the assistant detects errors in this experiment (i.e., the initial set of rules) is not influenced by the knowledge of the teachers participating in the experiment. In average, each work consisted of 16 classes and 520 lines of code. Although the projects were small, they involved checking at 7800 lines for a single teacher. The workload was divided between the two teachers: the first

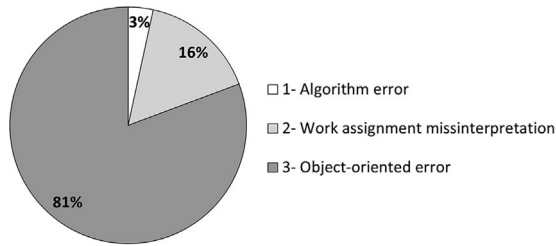


FIGURE 10 Teachers' corrections per category

teacher checked 8 works (about 4160 lines of code) and the second teacher checked the remaining 7 works (about 3640 lines of code).

The teachers made a total number of 88 corrections for the whole set of 15 courseworks. Those corrections were manually classified into three categories:

1. The "algorithm error" category: errors that arose when the program gives an unexpected output due to a programming bug.
2. The "work statement" category: Involve errors due to a misinterpretation of the work statement.
3. The "object-oriented" category: represents errors due to OOP misconceptions.

Figure 10 shows the distribution of the corrections made by teachers, according to the classification presented. There were just three algorithm errors (3%), which is expected since students already know basic programming structures. In addition, there were 14 work statement errors (16%), most of them related to the lack of functionality requested in the statement. Finally, there were 71 object-oriented errors (81%). Figure 11 shows the distribution of the teacher's corrections among the 15 student's works. This figure shows that all but one work presented some object-oriented error

(only the work #14 did not present this kind of error in the teacher's corrections).

With respect to the object-oriented errors (which are the focus of this work), Figure 12 shows the distribution of the 71 errors according to teachers' corrections. Each category corresponds to a specific type of object oriented error that is related to a student misconception.

The second part of the experiment consisted in checking the 15 students' works with Soploon. The experiment was carried out using the default set of 47 types of errors predefined in the assistant. These errors were previously defined based on other teachers' previous experience in the correction of exams and they include object-oriented errors commonly committed by novice programmers and program structures that usually reflect student misconceptions. During the experiment, the assistant detected 176 errors from only 21 different types of errors. Figure 13 details the number of errors detected by type of error. Notice that Figure 13 only shows the 21 types of errors detected by the assistant during the experiment (omitting the remaining 26 types of errors). A complete description of all the types of error that the assistant is capable of detecting (without the need of customization) is available in the Soploon website.⁹

Figure 13 sums up the errors detected by the tool highlighting how many times each type of error was detected. The light gray bar indicates the number of times an error was correctly detected. For example, the error of not abstracting a field was detected 43 times. Instead, the white bar indicates the number of times an error was incorrectly detected (i.e. the assistant detected an error, but it was not really an error). This may occur when the rule does not define specifically an object-oriented error, but it specifies a program structure that usually reflects student misconceptions. For example, during the experiment, this happened mostly with the detection of constants in code, since a constant in code not always means an error.



FIGURE 11 Teachers' corrections per student work

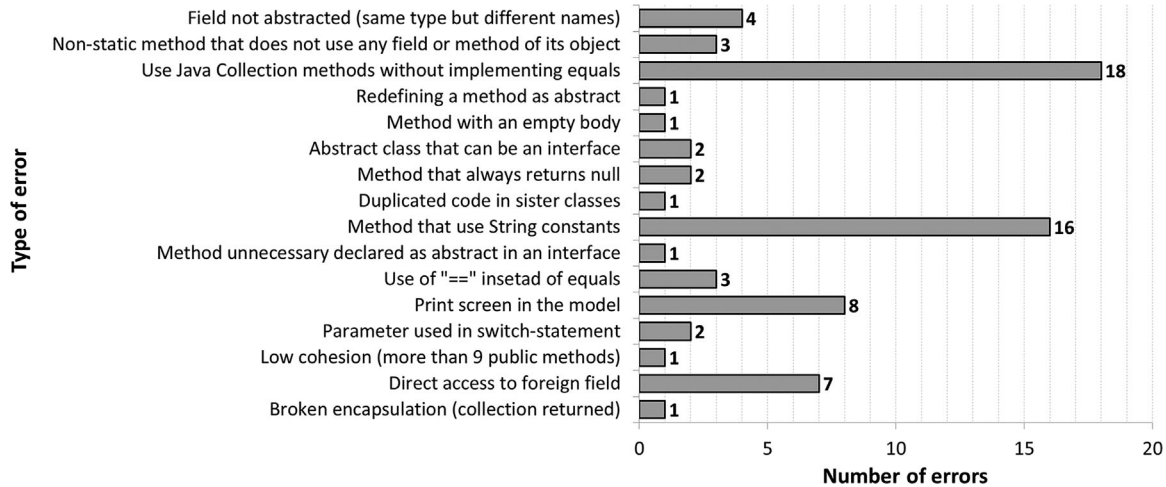


FIGURE 12 Object oriented errors detected by teachers

The errors detected by the assistant were compared with the teachers' corrections (Figure 14). The bi-color bars represent errors detected by the teacher but not detected by the assistant (note that there were no rules defined for detecting those errors). The dark gray bars indicate the number of errors detected by both the teachers and the assistant. The light gray bars represent the errors detected only by the assistant. The white bars represent the number of errors incorrectly detected by the tool. We can observe that by using the assistant, the teacher would have detected 196 object-oriented errors instead of 71 (i.e., 125 extra errors). With respect to the errors incorrectly detected by the tool, a teacher would

quickly discarded them after a simple manual check by taking advantage of the assistant's ability to locate and highlight any detected error in the source code.

After comparing the assistant corrections with respect to the teachers' corrections, we observed three types of object-oriented errors that the assistant was not able to detect (the three first errors of the plot in Figure 14). Those errors are: A non-static method that do not use any field or method of the object (it only uses parameters to compute something not related to the object itself); a non-abstracted field in sister classes (with the same type but different names); and using Java Collections methods (such as *contains* or *remove*)

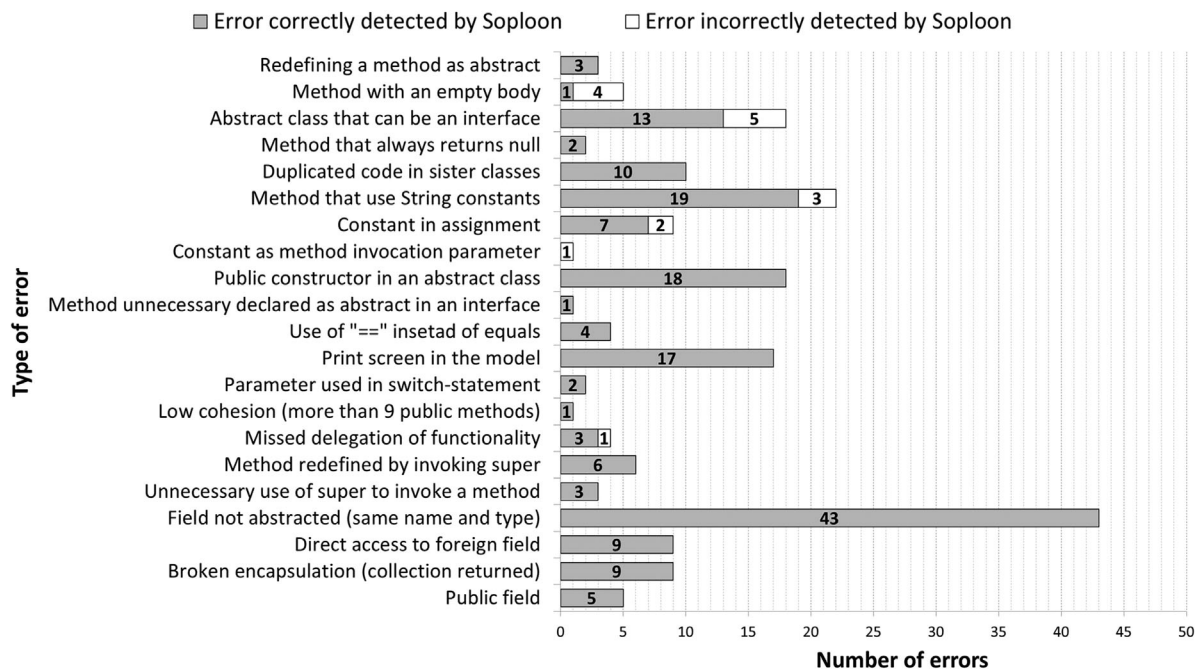


FIGURE 13 Object oriented errors detected by Soploon

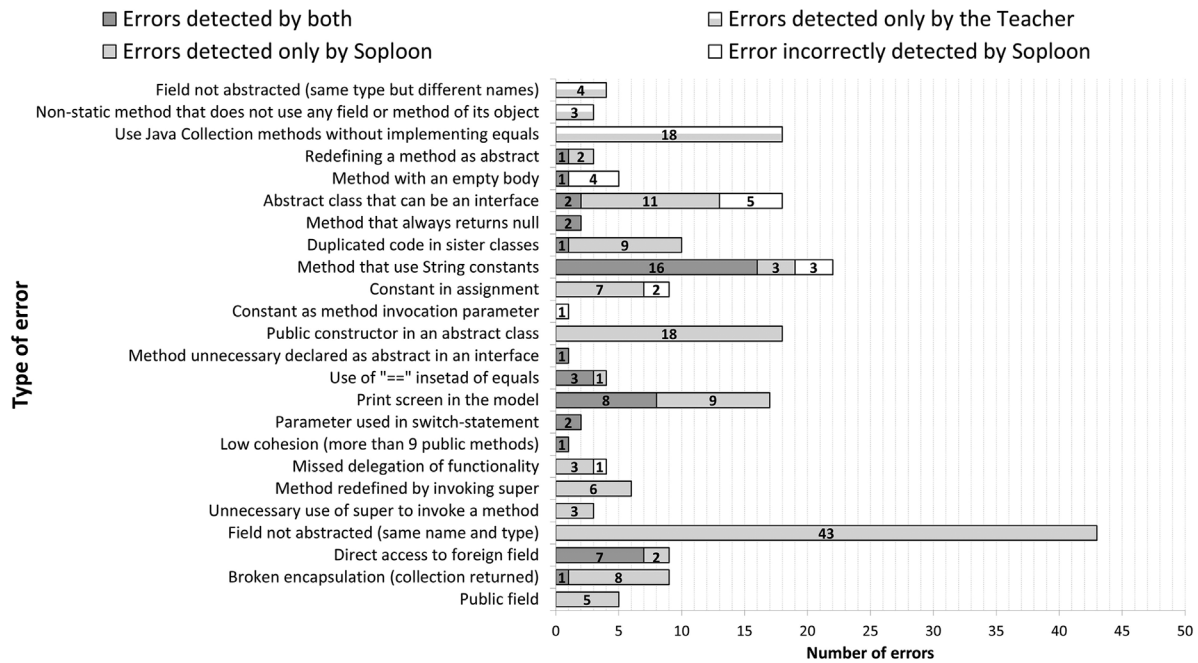


FIGURE 14 Object oriented errors detected by teachers and Soploon

without implementing the method *equals* in the objects stored in the collection; Those errors were added to the set of rules in order to customize the tool and check its capacity to detect new errors.

Once the three new rules were added, the second part of the experiment was repeated: the assistant checked the 15 student's works with the new extended set of Prolog rules. Figure 15 shows the results for the three new types of errors. After adding the three rules, the assistant was able to detect 29 errors overlooked by the teacher. Thus, the teacher does not need to exhaustively analyze each line of code, since Soploon can notify him/her when something may be wrong. The teacher just has to customize Soploon by adding new rules as he/she considers necessary.

To sum up our experiment, Figure 16 shows how the detection of errors for each student's work was improved by using the assistant. This Figure details the errors found by the teachers (the three initial series), and the errors detected by Soploon (the fourth serie). In this case, all the work's

corrections were improved, and even 22 object-oriented errors were detected in the work #14 (which initially has only one error, a work assignment misinterpretation). This shows the relevance of Soploon during the assessment task in order to give a more complete feedback to students.

6 | CONCLUSION AND FUTURE WORKS

In this work, we presented Soploon, an approach to detect object-oriented errors in Java source codes to help teachers in the task of assessing students' courseworks. Soploon is easily customizable to include new rules for detecting new errors or to modify the existing ones. We performed an experiment with real students to assess the efficiency of Soploon to detect errors, compared to real teachers.

Teachers can use Soploon during the assessment of students' source codes in order to make a more exhaustive

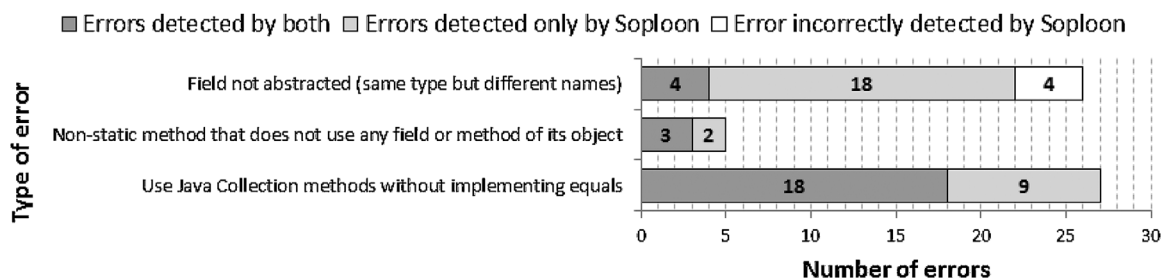


FIGURE 15 Object oriented errors detected by Soploon with the new rules

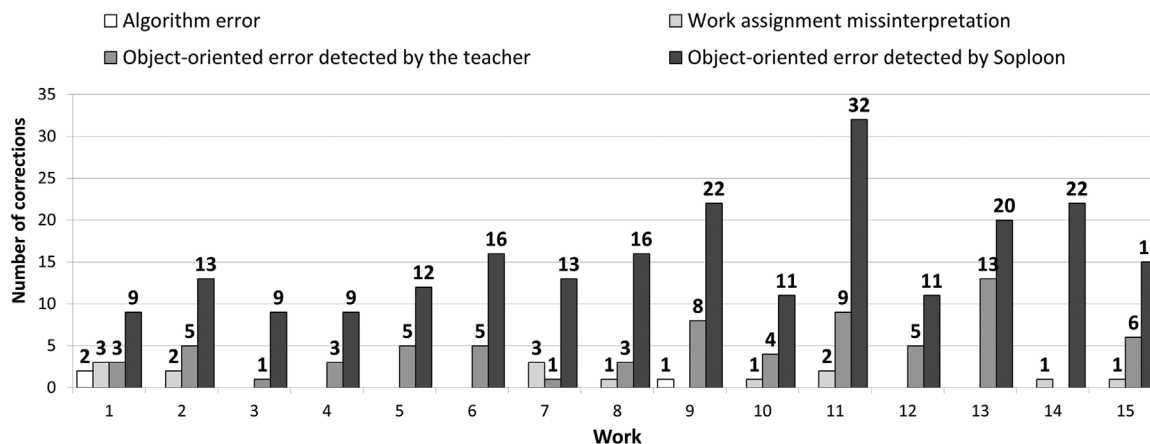


FIGURE 16 Corrections per student work

assessment and to avoid overlooking OOP errors or misconceptions. In this way, they can give a more complete and detailed feedback to students, and address students' misconceptions before they build wrong mental models about OOP. As an additional benefit we can mention that since by using Soploon, teachers workload is reduced, they can ask students to submit programming exercises more frequently. Consequently, teachers will be able to detect students' misconceptions earlier.

The first conclusion is that the assistant can help teachers to avoid overlooking major object-oriented errors. This is concluded from comparing teachers' corrections with respect to Soploon corrections. In these works, the assistant detected 125 object-oriented errors overlooked by the teachers as shown in Figure 14. However, it is important to note that the assistant may detect an error that does not actually exist. This may occur when a rule does not define an object-oriented error, but it specifies a program structure that usually reflects student misconceptions. In these cases, the teacher must manually check whether the program structure really represents an error or not. However, with the help of Soploon ability to locate and to highlight the detected errors in the source code, this is a simply task for a teacher.

The second conclusion is that teachers are able to detect errors that are impossible to be detected by the assistant. In the experiment, the teachers made several corrections related to object-oriented errors that were not included in the initial set of rules of the assistant. Obviously, these errors could not be detected by the assistant in a first instance. Then, after customizing the assistant and adding those errors to the data set of rules, the assistant was able to detect them. Moreover, other teachers' corrections were related to program bugs or to students' misinterpretation about the work statement. For example: the absence of required functionality, or an unexpected output for a given test case. The assistant cannot detect these errors since they are out of its scope.

To sum up, the experiment allows responding the initial questions:

- *Can the assistant detect errors overlooked by the teachers?* During the experiment, the assistant detected many errors overlooked by the teacher. This is helpful for giving a more complete and exhaustive feedback to students.
- *Can the assistant help to reduce teachers' workload?* By using the assistant, teachers can focus on analyzing the program design and the main classes, without the need of exhaustively analyzing each line of code. In this sense, the assistant reduces teachers' workload.
- *Can the assistant replace the teachers in the corrections of the works?* However, the assistant is not able to detect every kind of error. For example, the assistant is not able to detect errors that were not included in the set of rules of the assistant, program bugs, or errors originated by student's misinterpretations of the work statement.

As future work, we intend to customize the tool to assist students instead of teachers. As mentioned before, students prefer to learn by working and programming alone. However, during this process, there is no teacher to check students' programs and to detect students' misconceptions. In this context, Soploon could work as a virtual teacher to which the student can ask to check his/her source code. However, it is needed to be careful with the false positives of the assistant, since the student may not be able to realize that the assistant is detecting an error where there is none. This problem can be addressed by making the rules more strict (to reduce the number of false positives) and with a complementary explanation of each type of error. In this way, when an error is detected, the assistant can give the student an explanation about the error in order to help him/her to distinguish whether it is actually an error or not. Another option is to use the concept of warning instead of the concept of error. In this

sense, Soploon can warn students about something potentially wrong, and encourage them to ask their teachers about the problem. The objective of this future study is to evaluate if Soploon is able to reduce the students' steep learning curve by helping them to detect and fix their own misconceptions.

ENDNOTES

¹ <https://github.com/tsantalisi/JDeodorant>

² <https://pmd.github.io/>

³ <https://www.eclipse.org/>

⁴ <http://si.isistan.unicen.edu.ar/soploon/#/info/predicates>

⁵ http://si.isistan.unicen.edu.ar/soploon/#/info/auxiliary_predicates


⁶ <https://www.eclipse.org/jdt/>

⁷ <https://www.eclipse.org/jdt/ui/>

⁸ http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AS/index.html

⁹ <http://si.isistan.unicen.edu.ar/soploon>

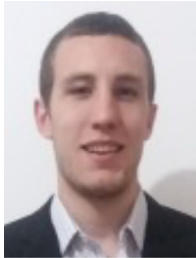
ORCID

Luis S. Berdun  <http://orcid.org/0000-0002-8006-8691>

REFERENCES

1. K. Bain, *What the best college teachers do*, Harvard University Press, Cambridge, 2004.
2. L. Berdun, A. Amandi, and M. Campo, *An intelligent tutor for teaching software design patterns*, *Comput. Appl. Eng. Educ.* **22** (2014), 583–592.
3. J. Biggs, *What the student does: Teaching for enhanced learning*, *High. Educ. Res. Dev.* **18** (1999), 57–75.
4. J. Bricchau and K. Mens, *Declarative meta programming* (Online), available online at: <http://soft.vub.ac.be/research/Old-DMP/> (Accessed: November 28, 2017).
5. W. K. Chen and Y. C. Cheng, *Teaching object-oriented programming laboratory with computer game programming*, *IEEE Trans. Educ.* **50** (2007), 197–203.
6. K. Craik, *The nature of explanation*, Cambridge University Press, Cambridge, 1943.
7. E. Denti, A. Omicini, and A. Ricci, *Multi-paradigm Java-Prolog integration in tuProlog*, *Sci. Comput. Program.* **57** (2005), 217–250.
8. M. Feldgen and O. Clua, *Games as a motivation for freshman students learn programming*, 34th Annual Frontiers in Education, 3 (2004), S1H/11-S1H/16.
9. F. A. Fontana, P. Braione, and M. Zanoni, *Automatic detection of bad smells in code: An experimental assessment*, *J. Object Technol.* **11** (2012), 1–38.
10. I. A. Halloun and D. Hestenes, *The initial knowledge state of college physics students*, *Am. J. Phys.* **53** (1985), 1043–1048.
11. M. Hristova et al., *Identifying and correcting Java programming errors for introductory computer science students*, Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, 2003, 153–156.
12. Y. Ito et al., *A method for detecting bad smells and ITS application to software engineering education*, 2014 IIAI 3rd International Conference on Advanced Applied Informatics, 2014, 670–675.
13. S. Kaur and R. Maini, *Analysis of various software metrics used to detect bad smells*, *Int. J. Eng. Sci. Technol.* **5** (2016), 15–19.
14. R. Kumar, J. Singh, and A. Kaur, *An empirical study of bad smell in code on maintenance effort*, *Int. J. Comput. Sci. Eng.* **5** (2016), 294–306.
15. E. Lahtinen, K. Ala-Mutka, and H. Järvinen, *A study of the difficulties of novice programmers*, *SIGCSE Bull.* **37** (2005), 14–18.
16. G. Licea et al., *Teaching object-oriented programming with AEIOU*, *Comput. Appl. Eng. Educ.* **22** (2014), 309–319.
17. C. Marinescu et al., *iPlasma: An integrated platform for quality assessment of object-oriented design*, Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005, 77–80.
18. S. Mekruksavanich and P. Muenchaisri, *Using declarative meta programming for design flaws detection in object-oriented software*, 2009 International Conference on Signal Processing Systems, 2009, 502–507.
19. N. Moha et al., *DECOR: A method for the specification and detection of code and design smells*, *IEEE Trans. Softw. Eng.* **36** (2010), 20–36.
20. N. Moha, Y. G. Gueheneuc, and P. Leduc, *Automatic generation of detection algorithms for design defects*, 21st IEEE/ACM International Conference on Automated Software Engineering, 2006, 297–300.
21. T. Muraki and M. Saeki, *Metrics for applying GOF design patterns in refactoring processes*, Proceedings of the 4th International Workshop on Principles of Software Evolution, 2001, 27–36.
22. Y. Qian and J. Lehman, *Students' misconceptions and other difficulties in introductory programming: A literature review*, *ACM Trans. Comput. Educ.*, **18** (2017), 1–24.
23. J. Rajesh and D. Janakiram, *JIAD: A tool to infer design patterns in refactoring*, Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming, 2004, 227–237.
24. A. Rao and N. K. Reddy, *Detecting bad smells in object oriented design using design change propagation probability matrix*, Proceedings of the International MultiConference of Engineers and Computer Scientists, **1** (2008).
25. S. R. Rist, *Teaching Eiffel as a first language*, *J. Object-Oriented Program.* **9** (1996), 30–41.
26. A. Robins, J. Rountree, and N. Rountree, *Learning and teaching programming: A review and discussion*, *Comput. Sci. Edu.* **13** (2003), 137–172.
27. N. Roperia, *JSmell: A Bad Smell detection tool for Java systems*, Long Beach, CA, USA: ProQuest Dissertation and Theses, 2009.
28. P. M. Sadler et al., *The influence of teachers' knowledge on student learning in middle school physical science classrooms*, *Am. Educ. Res. J.* **50** (2013), 1020–1049.
29. F. Simon, F. Steinbrückner, and C. Lewerentz, *Metrics based refactoring*, Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, 2001, 30–38.
30. P. Tan, C. Ting, and S. Ling, *Learning difficulties in programming courses: Undergraduates' perspective and perception*, Proceedings of the 2009 International Conference on Computer Technology and Development, **1** (2009), 42–46.

31. T. Tourwé and T. Mens, *Identifying refactoring opportunities using logic meta programming*, Proceedings of the 7th European Conference on Software Maintenance and Reengineering, 2003, 91–100.
32. A. Vihavainen, M. Paksula, and M. Luukkainen, *Extreme apprenticeship method in teaching programming for beginners*, Proceedings of the 42nd ACM technical symposium on Computer science education, 2011, 93–98.



S. VALLEJOS is a PhD student at UNICEN University, Argentina, with a research grant of CONICET (National Council for Scientific and Technological Research). He received a System Engineer degree in 2016 from the UNICEN. His research interests include natural language processing, social network analysis, and object-oriented programming.



L. S. BERDUN is a researcher at ISISTAN Research Institute (CONICET-UNICEN), Argentina, and a professor at UNICEN University, Argentina. He received a PhD degree in Computer Science in 2009 and a master's degree in Systems Engineering in 2005 from the UNICEN. His research interests include intelligent aided software engineering, planning algorithms, knowledge management.



M. G. ARMENTANO is a researcher at ISISTAN Research Institute (CONICET-UNICEN), Argentina, and a professor at UNICEN University, Argentina. He received a PhD degree in Computer Science in 2008 and a master's degree in Systems Engineering in 2006 from the UNICEN. His research interests include social network analysis, user modeling, and recommender systems.



Á. SORIA is a researcher at ISISTAN Research Institute (CONICET-UNICEN), Argentina, and a professor at UNICEN University, Argentina. He received a PhD degree in Computer Science in 2009. His research interests include software architectures, quality-driven design, object-oriented frameworks, and fault localization.



A. R. TEYSEYRE is a researcher at ISISTAN Research Institute (CONICET-UNICEN), Argentina, and a professor at UNICEN University, Argentina. He received a PhD degree in Computer Science in 2010 and a master's degree in Systems Engineering in 2001 from the UNICEN. His research interests include software visualization, software architecture, object-oriented frameworks and user centered design.

How to cite this article: Vallejos S, Berdun LS, Armentano MG, Soria Á, Teyseyre AR. Soploun: A virtual assistant to help teachers to detect object-oriented errors in students' source codes. *Comput Appl Eng Educ.* 2018;26:1279–1292.
<https://doi.org/10.1002/cae.22021>