Received June 30, 2020, accepted July 11, 2020, date of publication July 17, 2020, date of current version July 29, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3010172

Software Analytics to Support Students in Object-Oriented Programming Tasks: An Empirical Study

PASQUALE ARDIMENTO[1], MARIO LUCA BERNARDI[2], (Member, IEEE), AND MARTA CIMITILE[3], (Member, IEEE)
[1]Computer Science Department, University of Bari Aldo Moro, 70125 Bari, Italy
[2]Department of Engineering, University of Sannio, 82100 Benevento, Italy
[3]Department of Law and Economics, Unitelma Sapienza University, 00161 Rome, Italy

Corresponding author: Pasquale Ardimento (pasquale.ardimento@uniba.it)

ABSTRACT The computing education community has shown a long-time interest in how to analyze the Object-Oriented (OO) source code developed by students to provide them with useful formative tips. Instructors need to understand the student's difficulties to provide precise feedback on most frequent mistakes and to shape, design and effectively drive the course. This paper proposes and evaluates an approach allowing to analyze student's source code and to automatically generate feedback about the more common violations of the produced code. The approach is implemented through a cloud-based tool allowing to monitor how students use language constructs based on the analysis of the most common violations of the Object-Oriented paradigm in the student source code. Moreover, the tool supports the generation of reports about student's mistakes and misconceptions that can be used to improve the students' education. The paper reports the results of a quasi-experiment performed in a class of a CS1 course to investigate the effects of the provided reports in terms of coding ability (concerning the correctness and the quality of the produced source code). Results show that after the course the treatment group obtained higher scores and produced better source code than the control group following the feedback provided by the teachers.

INDEX TERMS Automated feedback, computing education, empirical experiment, object-oriented programming.

I. INTRODUCTION

Most of today's widely used programming languages (e.g. C++, Java, Python) follow an Object-Oriented Programming (OOP) paradigm that takes a primary role with respect to the traditional procedural languages. Consequently, since the beginning of the 2000s, a huge number of educational institutes recognize the importance to teach OOP in Computer Science courses [1], [2].

This allows the teacher to face several hurdles [3]–[5] due to the many factors ranging from difficulties within the subject (e.g. student's aptitude, multiple skills requirements) and methods (teaching strategies are quite new). This encourages continuous research of novel teaching strategies [6], [7] based on the investigation of students' hurdles to address student questions as soon as possible providing timely feedback [8], and improving retention [9].

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

According to this, several studies are focused on the comprehension of student mistakes [10] and misconceptions [11] to drive teachers. These code violations can be extracted and analyzed from samples of student source code and can be classified on the base of their relevant language constructs and some quality issues. This data, if adequately represented and analyzed, can be useful to understand how students head the development tasks, and to provide timely and useful feedbacks or batch reports.

Basing on the above considerations, this study proposes the Student Profiling Approach (SPA) aiming to analyze and report the most common violations of the OOP in the student source code. The approach is based on a violation model useful to verify the presence of a set of predefined violations in the students' source code giving useful feedback to the students during their coding activity. This model, differently from other existing approaches can be easily customized by the teacher on the base of specific educational goals. According to the above considerations, we have developed the so-called Student Profiling Tool (SPT) as an extension of the Eclipse Che cloud IDE, to analyze student's code and assess the quality of produced artifacts on the base of the proposed violation model. The environment implements a wide set of analyses of a different kind (depending on artifacts and the part of the software life-cycle understudy). In particular, it can detect the set of possible violations in student source code and produce individual reports of mistakes and related misconceptions. These reports convey useful feedbacks and formative tips to students about their coding activities, to improve the quality and the correctness of their work. This work presents an empirical study to investigate if continuous code analysis and reporting performed using SPA in a real course improve student's coding ability. Specifically, the study spans through six coding sessions involving students from a CS1 course [1] in the Computer Science Engineering degree held at the University of Bari. The results of the quasi-experiment highlight that the proposed approach significantly improves student's coding ability. Moreover, the produced reports also support instructors in designing and developing novel teaching strategies improving the effectiveness of Object-Oriented programming courses. In the next section, we summarize related work. Section III describes the basic notions for the feedback concept and the feedback model. Section IV explains the proposed approach for the source code analysis and student's report generation. Section V presents the experiment design whereas Section VI reports the obtained results. Section VII proposes an interpretation of the findings while in Section VIII the threats to validity are discussed. Section IX deals with conclusion.

II. RELATED WORK

Several studies investigated OO source code errors in the last years. Focusing on syntax [12], [13] some authors enhance compiler error messages [14]–[17], logic errors [18], [19], and coding habits [20]–[23]. Some other works are more focused on the OOP learning process presenting its main difficulties [11], [24], [25]. For example, authors in [19], analyze the code generated by the students (15,000 code fragments) and classified the student's logic errors as algorithmic errors, misinterpretations of the problem, and fundamental misconceptions. In [26], a survey about the aspects of the Java language perceived by the students as difficult is presented. Another approach [27] consists to examine the quality of student code with regards to program flow, functions, clarity of expressions, decomposition, and modularization. This study highlights that novice programmers usually use professional static analysis tools (Checkstyle,[1] PMD,[2] SpotBugs,[3] SonarQube.[4])

In [28], authors starting from the list of the problems detected by several professional tools, analyze the code of 3,691 students over five semesters. Basing on the above considerations, in [29], an error detection advisory tool named Expresso is introduced. It aims to help the teachers to understand the types of frequent errors the students make among a list of the typical logic, semantic and syntax errors usually made by novice programmers. Another approach [4] is based on the use of two checklists for grading student programs. These lists are obtained by considering basic OOP concepts and typical novice misconceptions as identified in the literature. The approach is used to evaluate objects-first CS1 course students.

Differently from the described approaches, our approach is more focusing on the teaching perspective allowing the instructor to customize the evaluation itself. In particular, the outcoming evaluation of the source code depends on the reference solution designed by the teacher (before each development task, the teacher has to carefully design and implement the reference solution for each assigned task). In this way, the tool is set on the base of the total number of mandatory properties and constraints that the solution must satisfy. Using this information it is established if a student committed a violation and, if so, to what extent it has been committed. Moreover, looking to the teacher's perspective, these approaches are not useful for preparing new teaching strategies since they are not based on an intuitive supporting environment to compare the results of student's code analyses [30]. In this paper, we present an approach to the student's code analysis able to detect their code violations. We exploit a supporting tool that can produce a detailed report on the violations of the source code given as input. These reports can be used to inform the student about their common Java language violations.

III. BACKGROUND

The proposed approach is focused on the concept of feedback model. In the past, several feedback models, based on educational theories, were proposed. In our approach, the feedback model is derived from the work of [31] later customized by [32] to the context of computing education and object-oriented programming tasks in particular. In the remaining of the section, the description of basic notions about such feedback models that are needed to understand the proposed approach is reported.

A. FEEDBACK CONCEPT

The importance of feedback for driving the student and handling its learning process and improving the learning outcome is largely discussed in literature [31], [32]. In particular, authors in [32] recognize the critical role of feedbacks in CS1/2 context to help learners improving coding abilities.

Feedback is defined in [31] as the information provided by an agent (e.g., teacher) about aspects of learner's performance or understanding. According to [31], to provide significant feedback is also necessary to address the following issues:

- *Type of feedback* — it consists of evaluative and informational components. The evaluative component relates

[1] http://checkstyle.sourceforge.net
[2] https://pmd.github.io
[3] https://spotbugs.github.io/
[4] http://www.sonarqube.org

to the learning outcome and indicates the performance level achieved. The informational component, instead, consists of additional information relating to the topic, the task, errors, or solutions. Combining the evaluation and information component of feedback might result in a large variety of feedback contents.

- *Technique* — it refers to what happens inside a tool to generate feedback.
- *Input type* — it refers to an input type that enables teachers to create exercises and influence the generated feedback by a tool. The most input type used is: *solution templates* presented to students for didactic or practical purposes as opposed to technical reasons such as easily running the program; *model solutions* that are correct solutions to a programming exercise; *test data*, by specifying program output or defining test cases; *error data* such as bug libraries, buggy solutions, buggy rules, and corresponding correction rules.
- *Timing of feedback* — it refers to the time when feedback is provided. This issue is extremely important as demonstrated in [33].

B. FEEDBACK MODEL

The proposed approach is based on the model of feedback as described in [32]. This study introduces contemporary models of effective feedback practice offering an interpretation of those in the CS1 context. Respect to the inspiring model, proposed by Hattie and Timperley [31], in [32] authors focus on aspects that have a widely recognized effect on learning in CS1 context. Specifically, they consider different levels of feedback corresponding to different learning aspects:

- **Task level** — This level includes feedback about "how well a task is being accomplished or performed, such as distinguishing correct from incorrect answers, acquiring more or different information, and building more surface knowledge" [31]. This type of feedback can be conceived along with several dimensions, such as high to low complexity, individual or group performance, and written or numeric notations. At this level, the literature suggests simplicity: more are simple the tasks more the task performance benefits from the feedback [34], more are simple the provided feedback more the feedback tends to be effective. Moreover, it is preferable to deliver and receive feedback in an individual situation than in a group situation. Feedback messages delivered in groups may be confounded by the perceptions of relevance to oneself or other group members. Finally, feedback at this level is more effective when students feel to be committed and involved in the task perceiving the impact on their performance.
- **Process level** — This feedback clarifies the processes necessary for task completion. At this level two aspects are important: i) helping students to develop strategies for error detection and correction, ii) providing students with cues and hints to guide their search for relevant information, and to apply effectively useful strategies. The main usefulness of this feedback is to suggest students the more effective and alternative solution strategies as pointed out by [31]. Feedback information about the processes underlying a task also can act as a cueing mechanism and lead to more effective information search and use of task strategies. Cues, to be more useful, should assist students in rejecting erroneous hypotheses and provide direction for searching and strategizing.
- **Self-regulation level** — Feedback at the self-regulation level is aimed to improve students' self-monitoring skills and to address the way students direct and regulate their learning. It comprises several aspects such as the capability to create internal feedback and to self-assess, the willingness to invest effort into seeking and dealing with feedback information, the degree of confidence or certainty in the correctness of the response, the attributions about success or failure, and the level of proficiency at seeking help.

As better specified in the following section, to improve learning, all these components should be considered when providing customized reports for students. From the detected student's mistakes, our feedback model derives the related misconceptions about object-oriented concepts and, to mitigate them, acts simultaneously at task, process and self-regulation level providing, respectively, a correct solution, proper guidance towards one or more correct solutions and a set of suggested readings strictly related to the detected misconceptions.

IV. STUDENT PROFILING APPROACH

The proposed approach is based on a cloud IDE able to analyze student's source code, identify their source code violations and providing them feedback with tailored reports. According to the feedback framework reported in the background section, the proposed approach carefully defines the adopted feedback model. Specifically, the considered mistakes and their underlying misconceptions are described in Section IV-A whereas the structure and feedback generation issues are addressed in the tool description (Section IV-B). Finally, to answer the input type issue we introduce and describe our violations model (it represents the model solution) and the adopted feedback reports (solutions templates) respectively in Section IV-A and Section IV-C.

A. THE VIOLATIONS MODEL

To verify the presence of mistakes in the student's source code our approach is based on a violations model. The model can be defined by the instructor and consists of a list of possible violations stored in a violation repository as source code static and dynamic analysis modules. The adopted model is obtained identifying for the core topics (as defined in [1]), the corresponding possible violations on the base of instructor experience (derived by manual student's source code analysis) and literature analysis [3], [5]. In particular, we started from such misconceptions of novice

TABLE 1. The violation model for source code correctness and quality assessment.

Category	Violations group	Violation	ID	normalized	experimentation
Foundation	State Design	Class Without Instance Fields	cwif	-	✓
		Missed Constant	mc	-	✓
		Missing Attribute	ma	✓	✓
	Behaviour Design	Class With Implicit Constructor	cwic	✓	-
		Public Field Changed by private methods	pfc	✓	-
		Missing Operation	mo	✓	✓
		Unused Association	uc	✓	-
	Abstraction	Empty NOn-ABstract method in root class	enoab	✓	-
		Missed Abstract Class	mac	✓	✓
	Hierarchy	Inheritance to Extend Values	iev	✓	-
		Misused Constructor Chaining	cc	✓	-
		Missed Generalization Relationship	mgr	✓	✓
		Missed Aggregation Relationship	mar	✓	✓
	Typing	Missed Inclusion Polymorphism	mip	✓	✓
		Missed Parametric Polymorphism	mpp	✓	-
		Wrong Use of Bounded Type Parameter	wubtp	✓	✓
	Encapsulation	Poor Interface usages and definitions	pi	-	✓
		Unused Private Method	upm	✓	-
		Wrong Visibility of Class Member	wvcm	✓	✓
		Misused Default Package	dp	-	✓
Quality	Bad code smells	Field Used as Local Variable	fulv	✓	-
		Local Variable Shadowing a Field	lvsf	✓	-
		Static Invocation Through Instance	siti	✓	-
		Unnecessary Object Casting	oc	-	✓
		Unused Private Field	upf	-	✓
		Cloned Code	clc	-	-
		Long Parameter List	lpl	-	-
		God Class	gc	-	-

programmers as described in the literature to find how these cause violations of language constructs. Successively, similarly to Sanders and Thomas [4], we manually inspected a sample of student programs (namely 162 of 1627 Java projects extracted from Blackbox [35], [36]) to identify and define the violations list as reported in Table 1. The table shows the groups of considered violations, the single violations description, and its corresponding acronyms. In the following subsections, we explain the listed violations reporting the related literature.

We categorized all violations from literature with respect to two main broad categories: the object-oriented language core topics (i.e., design, abstraction, hierarchy, typing, and encapsulation) and quality-related metrics (we considered those known in the literature as bad code smells).

1) STATE DESIGN

Object-oriented design is meant as decomposition into objects carrying state and having behavior [1] where the state of an object encompasses all of the properties of the object plus the current values of each of these properties [37]. According to this, we identified the following state design violations:

- **Class Without Instance Fields (cwif)**: It is well-known that students have difficulties in understanding the concepts of object and class [11]. A class without instance fields (*cwif*) could be a consequence of such difficulties. Of course, a class without instance fields containing only static methods (like the entry point — i.e. the `main` method) is not considered a violation.
- **Missed Constant (mc)**: A class field which is only read should be declared as constant. This is in line with Chen *et al.* [38] who discovered misconceptions when students determine which data member is appropriate for declaring a constant. Moreover, Ragonis and Ben-Ari [11] list difficulties in understanding the static aspect of the class definition.
- **Missing Attribute (ma)**: An attribute is a named property of a class that describes a value held by each object of the class. Sometimes could happen that a property of a class is not modeled. In this case, the class does not accomplish the requirement related to the missing property.

2) BEHAVIOUR DESIGN

The behavior of an object is how an object acts and reacts, in terms of its state changes and message passing [37]. We defined the following behavior design violations:

- **Class With Implicit Constructor (cwic)**: Ragonis and Ben-Ari [39] consider teaching constructors a difficult multiple choice and they found that the professional style of declaring a constructor to initialize attributes from parameters is to be preferred even though it seems difficult to learn. This means that a class with an implicit constructor (*cwic*), listed among simpler styles, should be avoided.
- **Public Field Changed by private methods (pfc)**: The difficulties to understand the influence of method execution on the object state is another problem related to the concepts of object and class [11].

- **Missing Operation (mo)**: An operation denotes a service that a class offers to its clients [37] corresponding to a single specific functional requirement. Sometimes the service of a class might be not modeled. In this case, the class does not accomplish the requirement related to the missing service.
- **Unused association (ua)**: Writing of a program that includes composed classes is a complex task [40]. Such difficulty could have led to including associations which are never used (i.e., the relationship attribute is never accessed and no message is sent to the linked object).

3) ABSTRACTION VIOLATIONS

Abstraction refers to the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer [37]. This group includes the following violations:

- **Empty NOn-ABstract method in root class (enoab)**: When a method of a root class in a hierarchy is intended to be specified by sub-classes it should be marked as abstract forcing the sub-classes to provide an implementation for it [41]. In addition, it could be made *final* to express that its behavior must not be overridden. Conversely, if a method is intended to be overridden, the intent fails to be expressed because sub-classes are free to not override it. In this case, it should be declared as abstract (leaving its definition to sub-classes).
- **Missed Abstract Class (mac)**: An abstract class is a class that cannot be instantiated because is incomplete. Its subclass will add to the structure and the behavior, usually by providing the implementation of incomplete parts. Often novices miss even the existence of abstract classes being unable to identify a common structure and to assign specific responsibilities to sub-classes. This violation is able, using source code analysis, to identify such missed factorization opportunity highlighting the code reduction of its application to developers.

4) HIERARCHY VIOLATIONS

Hierarchy could be defined as a ranking or ordering of abstractions. [37]. In a complex system, two kinds of hierarchies interact in several ways: class hierarchies, that are based on specialization (i.e., "is a" relationships), and the object structure hierarchies, that are based on aggregation and composition (i.e., "part of" relationships). The following violations reflect known misuses of such constructs that do not depend on the particular semantics of the modeling context and can be detected automatically by source code analysis:

- **Inheritance to Extend Values (iev)**: Inheritance can be mistakenly used to extend values [25]. For instance, some students think that inheritance can be used to change the values of fields, rather than for adding attributes or operations.
- **Misused Constructor Chaining (cc)**: The study in [25] also highlights how many students fail to understand the chain of constructor calls in object creation. Even though the choice of using the default constructor was deliberate, it is always worth giving feedback to students to avoid the proliferation of (bad) long-term habits—as suggested by Ala-Mutka [42].
- **Missed Generalization (mg)**: Generalization and inheritance are useful concepts that help to reduce the complexity of models and the redundancy in specifications. Moreover, they increase the reuse of specifications and code [43]. This could also imply the presence of duplicate code.
- **Missed Aggregation (ma)**:
  Aggregation is used to model whole-part relationships in which multiple classes are combined to generate a class with a more complex internal structure. Often, inexperience leads to the use of static relationships such as inheritance instead of aggregation and this produces less flexible systems with more classes. The use of aggregation and inheritance should always be linked to the existence of whole-part relations and specialization (i.e., "is-a") avoiding misuse.

5) ENCAPSULATION

Encapsulation consists of compartmentalizing the elements of an abstraction that constitute its structure and behavior. It serves to separate the contractual interface of an abstraction and its implementation [37].

Considered encapsulation violations are:

- **Poor interface use and def (pi)**: Among the good coding practices for Java, some studies [44] advise to prefer the use of interface types (over class types) for all declared types. In other words, a declared site (e.g. a local declaration) should use an interface (when such an interface is available).
- **Unused Private Method (upm)**: Private methods which are never called introduce dead code into the system. This is also linked with difficulties with scope, namely with the "private" keyword.
- **Wrong Visibility of Class Member (wvcm)**: Because public fields tend to limit flexibility in changing source code use, a good principle is to avoid public fields except for constants.
- **Misused Default Package (dp)**: Default or unnamed packages are provided by the Java platform principally for convenience when developing small or temporary applications or when just beginning development. However, it is very important to learn since the beginning to use packages otherwise any other class in the default package has access to class fields and methods that are not set to private.

6) TYPING

Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged [37].

Polymorphism represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass. Any object denoted by this name is, therefore, able to respond to some common set of operations. Some typing violations are:

- **Missed Inclusion Polymorphism (MIP)**: In inclusion polymorphism, an object can be viewed as belonging to many different classes which need not be disjoint, i.e. there may be the inclusion of classes.
- **Missed Parametric Polymorphism (mpp)**: Parametric polymorphism is obtained when a function works uniformly on a range of types: these types normally exhibit some common structure.
- **Wrong Use of Bounded Type Parameter (wubtp)**: The Bounded type parameter allows restricting the types that can be used as type arguments in a parameterized type. Students have issues in understanding how to bound the parameterized type leading to subtle bugs at run-time. SPT reports highlight these situations clarifying the consequences of dangerous type parameter management.

7) BAD CODE SMELLS

Bad code smell is a surface indication that usually corresponds to a deeper problem in the system [45].

The considered code smells violations are:

- **Field Used as Local Variable (fulv)**: Students have issues in understanding the difference between class fields and local variables inside methods Tempero [46].
- **Local variable shadowing a field (lvsf)**: The shadowing of a field by the definition of a local variable with the same name is related to the same motivations of *fulv*.
- **Static Invocation Through Instance (siti)**: Static methods should be accessed in "a static way". When a static method is accessed by an instance, this probably represents a warning sign of an incorrect comprehension of the meaning of static methods. It is important to understand the current object and its usage [47].
- **Unnecessary Object casting (oc)**: Another issue with polymorphism is using typecasting explicitly (especially when is not needed at all) [48].
  Typechecking bad code smells are also a bad practice and usually programmers who have not fully understood the object-oriented paradigm uses conditional statements to simulate dynamic dispatch and late binding [49].
- **Unused Private Field (upf)**: It refers to a situation with a private field not assigned and not used at all. Private fields that are not used at all are a form of dead code and should be removed. One exception to this violation is the Java serialization where a serializable class can declare its own *serialVersionUID* explicitly by declaring a field named *serialVersionUID*.
- **Cloned Code (clc)**: It means that identical or very similar code exists in more than one location making source code more difficult to maintain.
- **Long Parameter List (lpl)**: A long parameter list can indicate that a new structure should be created to wrap the numerous parameters or that the method is doing too many things. Typically a method suffers from lpl when has four or more parameters.
- **God Class (gc)**: The *God class* is a kind of class that knows too much or does too much. That means a huge class in terms of the number of lines of code, creating tight coupling and increasing the challenges in the source code maintainability.

B. STUDENT PROFILING TOOL

The proposed approach uses a cloud-based reengineering version of our previous Student Profiling Tool (SPT) [50] that was standalone. It is based on the Eclipse Che platform to add both real-time learning analytics features and the support to handle monitoring of labs and development session recording the performances of students and developers (in terms of correctness). The overall architecture of SPT with its main components is represented in Figure 1. The main components are the Violation Model Manager and Assignments Manager. The Violation Model Manager allows the instructor to specify the violation model that is enforced when analyzing student's projects. Its presence distinguishes our tool from the 96% of existing tools providing feedback on "knowledge about mistakes" [51]. SPT, instead, provides feedback based on "knowledge about concepts", i.e. the Object-Oriented concepts. It is based on a Basic static analysis (BSA) technique. BSA, in general, analyses a program (source code or bytecode) without running it, and can be used to detect misunderstood concepts, the absence or presence of certain code structures, and to give hints on fixing these mistakes. In our case, instead, this technique is used to analyse source code developed by the student with the model solution provided by the instructor (model solution template). If in the source code the tool detects any violation, defined in the violation model, gives hint on how to fix this violation. The Assignments Manager allows the instructor to define assignments linking them to the solutions produced by students (collected in a Project Data Store). These projects are subsequently analyzed to recover the errors and mistakes following a predefined violation model (Violations Repository).

The results of static analysis are managed by the Assignment Manager component and will be summarized on the client-side by the report builder for both instructors and students basing on their role. The Report Builder performs data aggregation and integration with existing software thanks to a custom Visualization Engine which supports different types of output (raw text, HTML, Latex, PDF or ad-hoc visualizations).

In addition to these standard components, SPT architecture includes a real-time reporting tool used to provide feedback and formative tips during development activities.

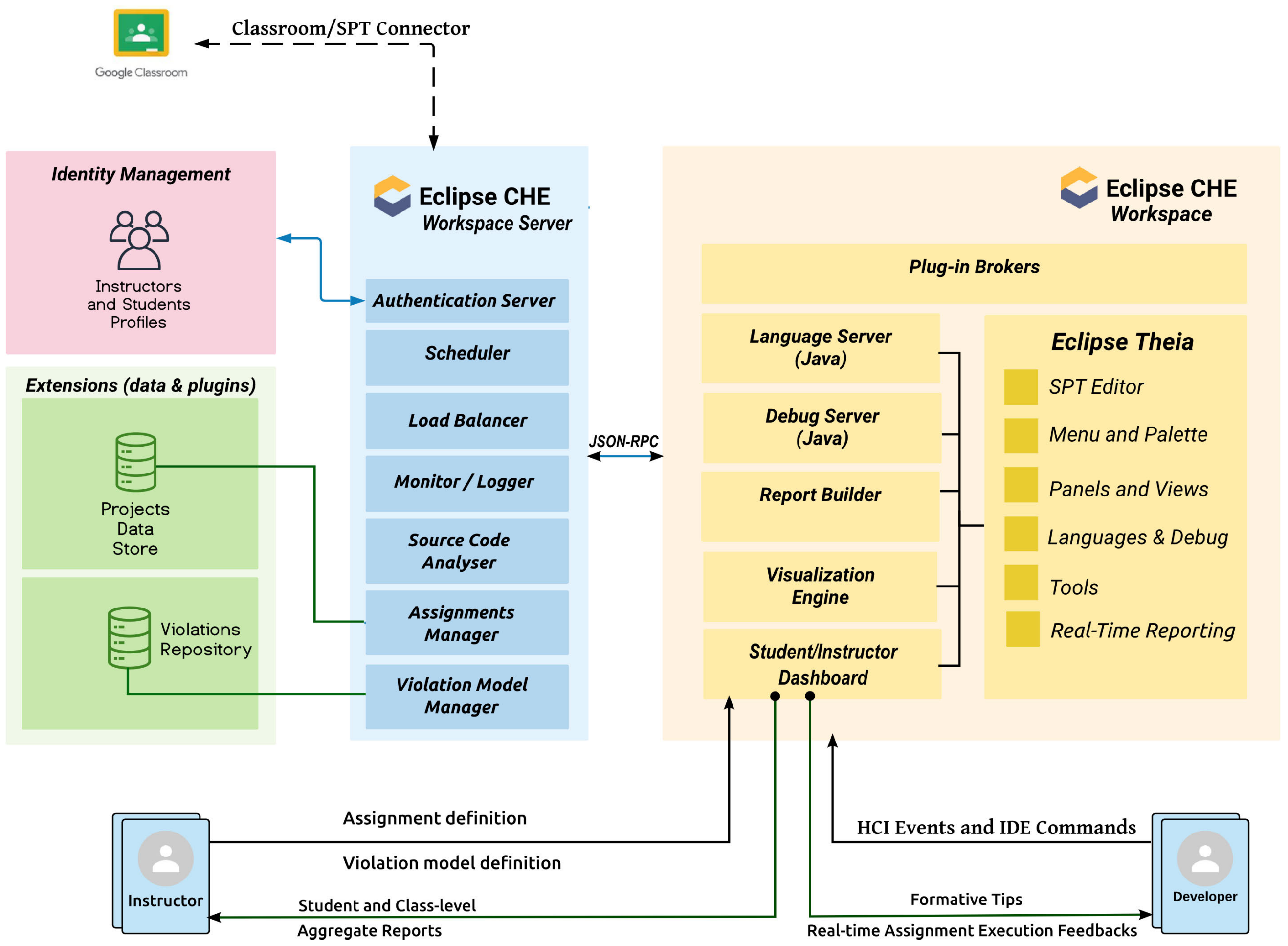


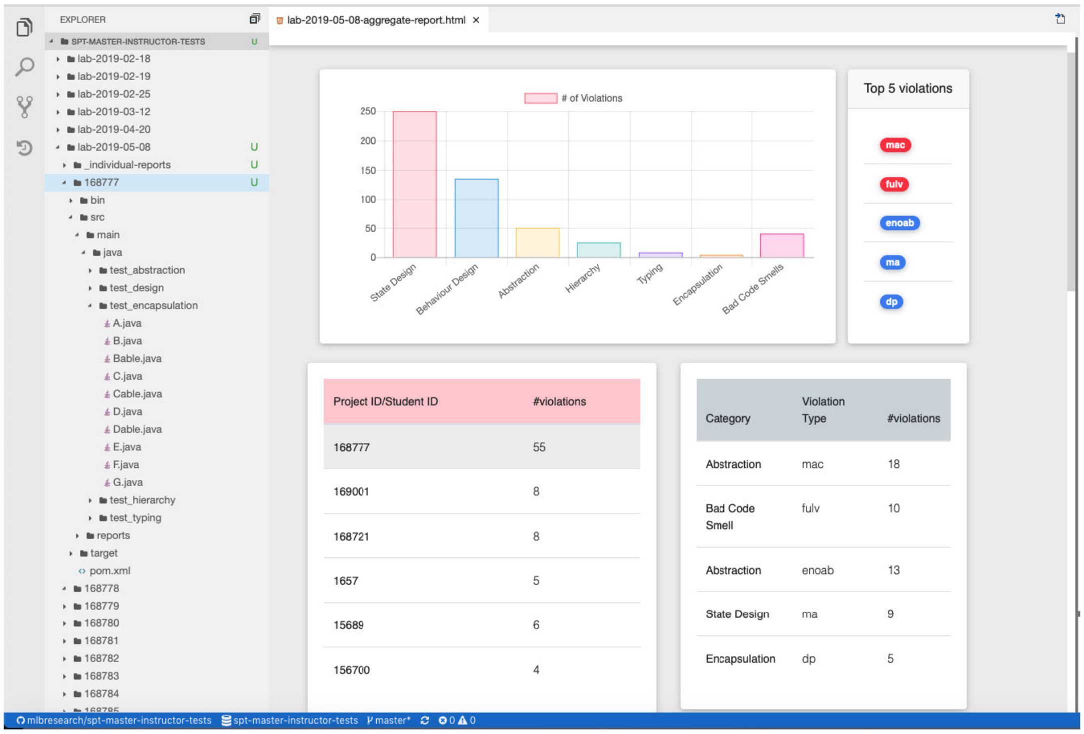
FIGURE 1. Student profiling tool architecture (extending Eclipse Che).

Figure 2 shows an example of the typical SPT session from the instructor perspective (student identity and other information are pseudonymized using generated UUIDs in the reports). The instructor has access to each lab in which a class has been involved SPT allows the generation of aggregate reports like the one shown in the central portion of the figure in which global statistics concerning the violations for the entire class are generated. This allows the instructor to track and follow the progress of the entire class. The aggregate report is navigable and can be inspected to see, for each category of violations, what are the most frequent ones, the top violations, and other useful aggregate measurements. Instructors can focus on particular project execution (i.e. student executing an assignment) by clicking on the ID in the report. SPT, in this case, provides detailed information on the violation committed. Moreover, the number of violations per student and its breakdown details (bottom right, i.e. "Category|Violation Type|#violations") are described as interactive tables.

Figure 3 shows, instead, an example of the typical SPT session from the student perspective (also in this case student identity and other information are pseudonymized using generated UUIDs in the reports). When a student, logs into his SPT account, he is recognized by the environment, traced and the produced project is stored and shown in the explorer view on the left side. On the right side, instead, the aggregate report concerning the violations he committed is reported (detailedreport.html in Figure 3). For instance, a student whose *id* is "168777", can access to details about the mistakes he performed in the session by clicking on a particular violation type in the right table, (`pi` for example). The student can visualize the description along with specific information for each violation ("Class|Related interface|Line" in case of `pi`), shown in Figure 3. This view is the typical student dashboard that allows students to trace errors and mistakes directly to their code by clicking on them. Specifically, by choosing a row like "A", Java code (with line numbers, syntax highlighting and injected comment) is presented with a marker on the line when violations occur. To know the committed violation and how to address it, the student has the possibility, to access a report (one per each violation committed).

C. THE FEEDBACK REPORTS

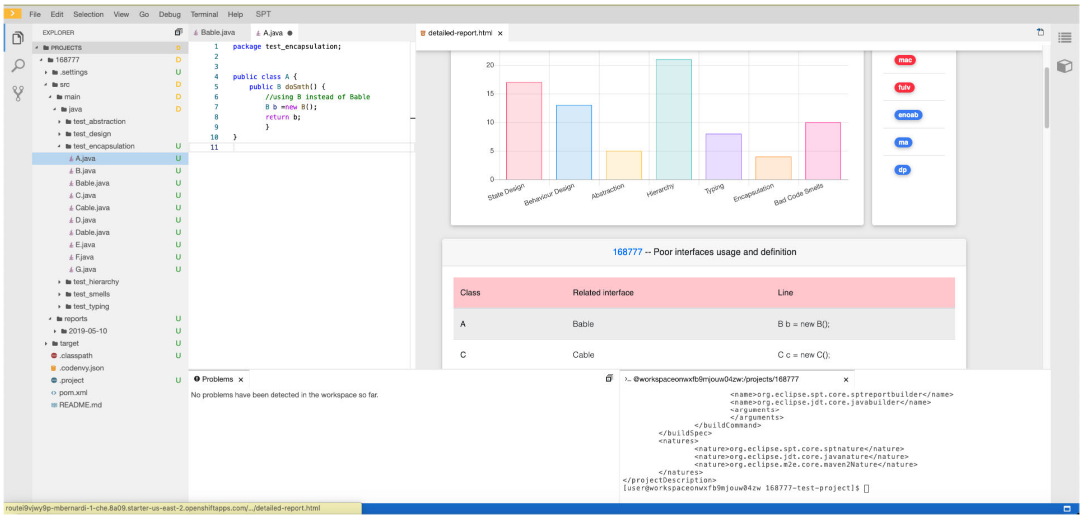
The approach requires a cooperation between the tool and the instructor. The instructor, for a given assignment, needs to write a model solution allowing the tool to compare students'



Project ID/Student ID	#violations
168777	55
169001	8
168721	8
1657	5
15689	6
156700	4

Category	Violation Type	#violations
Abstraction	mac	18
Bad Code Smell	fulv	10
Abstraction	enoab	13
State Design	ma	9
Encapsulation	dp	5

FIGURE 2. SPT instructor dashboard: it provides navigable reports with aggregate statistics evaluated for all the projects of a lab and allows the management of individual reports for students.



```
package test_encapsulation;


public class A {
    public B doSmth() {
        //using B instead of Bable
        B b =new B();
        return b;
        }
}
```

168777 -- Poor interfaces usage and definition

Class	Related interface	Line
A	Bable	B b = new B();
C	Cable	C c = new C();

No problems have been detected in the workspace so far.

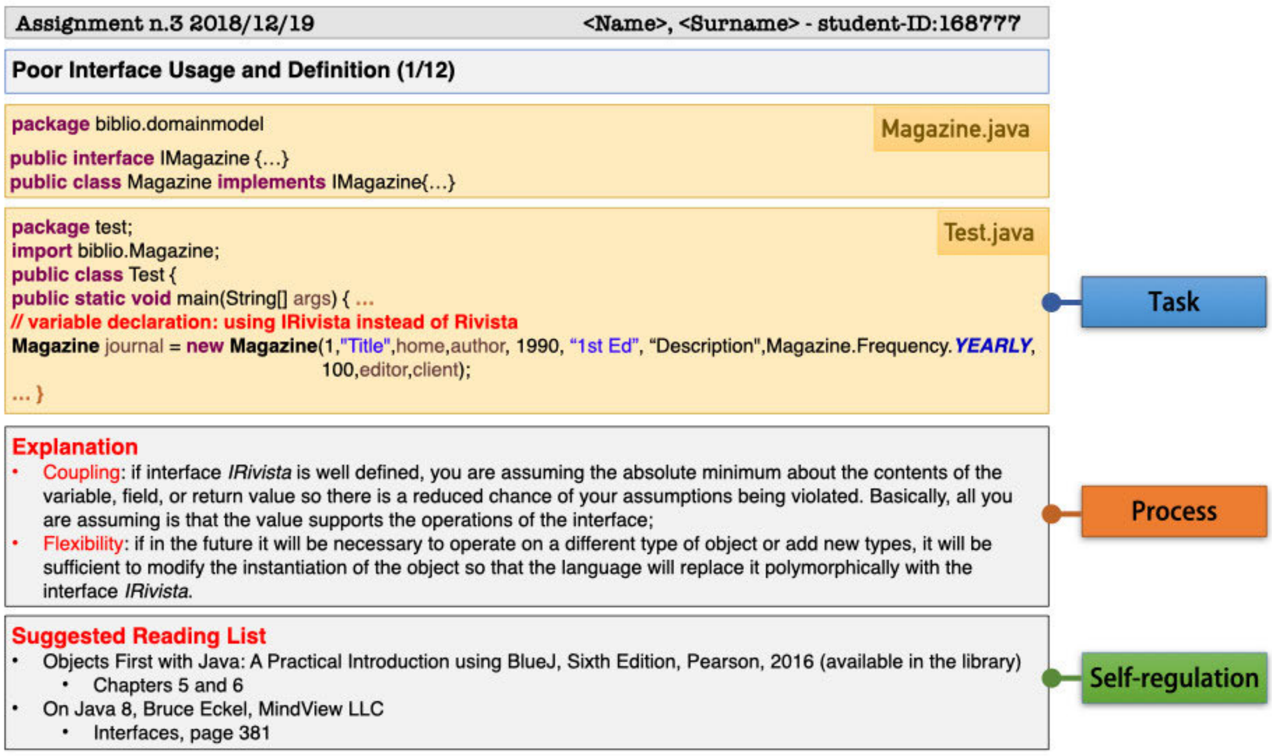
```
>_ @workspaceonwxfb9mjouw04zw:/projects/168777
                        <name>org.eclipse.spt.core.sptreportbuilder</name>
                        <name>org.eclipse.jdt.core.javabuilder</name>
                        <arguments>
                        </arguments>
                </buildCommand>
        </buildSpec>
        <natures>
                <nature>org.eclipse.spt.core.sptnature</nature>
                <nature>org.eclipse.jdt.core.javanature</nature>
                <nature>org.eclipse.m2e.core.maven2Nature</nature>
        </natures>
</projectDescription>
[user@workspaceonwxfb9mjouw04zw 168777-test-project]$ 
```

routei9vjwy9p-mbernardi-1-che.8a09.starter-us-east-2.openshiftapps.com/.../detailed-report.html

FIGURE 3. SPT student dashboard: it provides both real-time and batch navigable reports (depending on instructor settings) for specific projects owned by the student.

solution against it (enabling also semantic violations detection). SPT checks the suite of violations against students' source code and suggests, using the model solution as a gold standard, a report based on detected problems. Since the presence of semantic violations does not necessarily imply a bad solution (i.e., the student's solution could be still correct but different from the instructor's one) SPT allows the instructor to confirm or to delete issues to be included in the final report. This also means that if the model solution is poorly defined the approach performs badly since is following wrong directives. In our experiment, to avoid bias of any kind, model solutions were prepared by a small team of expert instructors and double-checked by another instructor that did not participate in solution definition. Moreover, to overcome the problem of timeliness, the feedback was provided at the end of each assignment task. Because each task distances one week from the next one, each student had sufficient time to understand feedback and try to improve his/her solution.

The feedback report is achieved from the considered violations model and provides students with support at various levels as described in Section III-B.


Assignment n.3 2018/12/19 <Name>, <Surname> - student-ID:168777

Poor Interface Usage and Definition (1/12)

```
Magazine.java
package biblio.domainmodel
public interface IMagazine {...}
public class Magazine implements IMagazine{...}
```

```
Test.java
package test;
import biblio.Magazine;
public class Test {
public static void main(String[] args) { ...
// variable declaration: using IRivista instead of Rivista
Magazine journal = new Magazine(1,"Title",home,author, 1990, "1st Ed", "Description",Magazine.Frequency.YEARLY,
                                    100,editor,client);
... }
```

Explanation
- Coupling: if interface *IRivista* is well defined, you are assuming the absolute minimum about the contents of the variable, field, or return value so there is a reduced chance of your assumptions being violated. Basically, all you are assuming is that the value supports the operations of the interface;
- Flexibility: if in the future it will be necessary to operate on a different type of object or add new types, it will be sufficient to modify the instantiation of the object so that the language will replace it polymorphically with the interface *IRivista*.

Suggested Reading List
- Objects First with Java: A Practical Introduction using BlueJ, Sixth Edition, Pearson, 2016 (available in the library)
  - Chapters 5 and 6
- On Java 8, Bruce Eckel, MindView LLC
  - Interfaces, page 381

FIGURE 4. Sample report regarding pi violations.

Concerning to the task level, the proposed violations model allows identifying the student's errors affecting the proposed solution. Concerning the process level, our feedback shows the right strategy to reach a correct solution. Finally, according to the self-regulation level, the feedback shows the list of suggested reading to help students to mitigate misconception and to reason critically about their own weakness and misunderstanding. An example of a report is shown in Figure 4, it refers to a pi violation. Each report consists of five fields:

- student's name, surname and id;
- the name of violation ("Poor Interfaces Usage and Definition" in the example);
- an excerpt of class code where the violation has been made (class Test in the example) and, whenever possible, the classes conceptually linked to the violation (IRivista and Rivista);
- the in-line code comment, highlighted in brown color, explaining how to address the violation ("variable declaration: using IRivista instead of Rivista" in the example);
- the explanation of why the source code should be written in a different way. In the example, it is explained why the use of an interface, instead of a concrete class, makes more flexible the source code produced;
- a list of suggested readings to understand why a violation has been committed. Usually, it includes lecture notes, open-source books, specific books with chapters, pages, and samples (e.g. Google books links). In the example, two chapters of a book and a specific page of another book are suggested.

In the figure, we highlight with colored labels the correspondence between the report field and the feedback level.

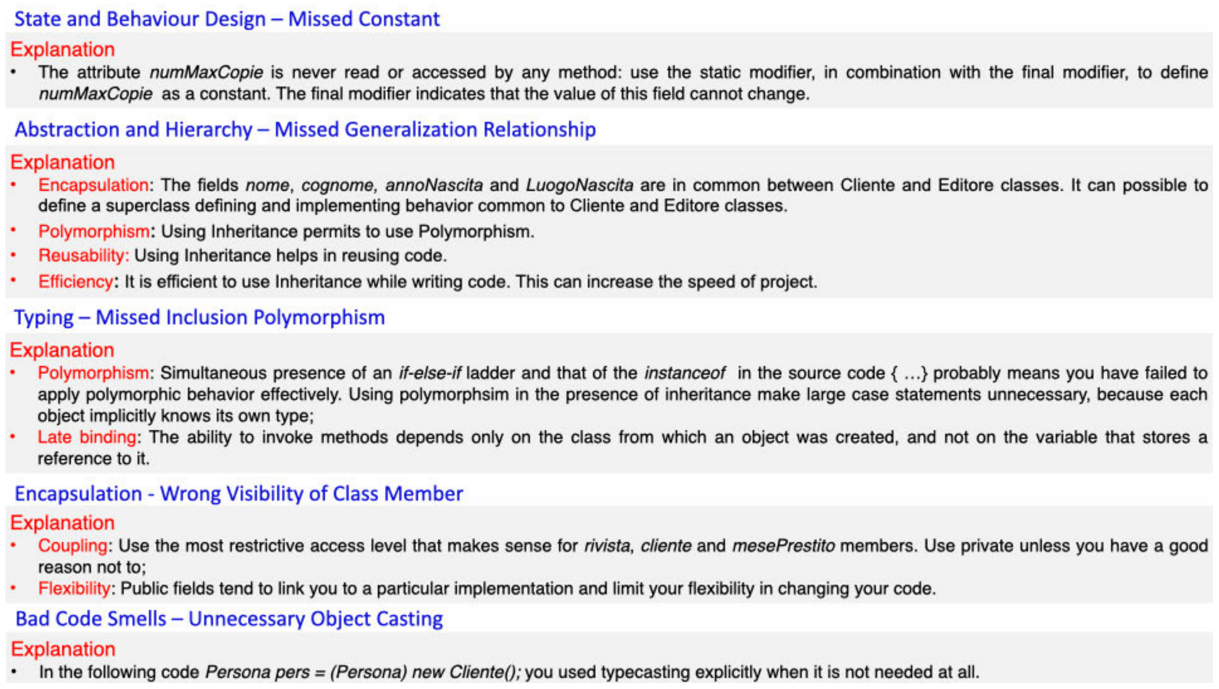
Figure 5 shows excerpts of several reports grouped by broad categories; a label placed above each excerpt indicates both the category and the violation which the explanation refers to. The excerpts report only the explanation section.

V. EMPIRICAL EVALUATION

The current empirical research was carried out as a quasi-experiment, since it was not possible to assign the treatments to the subjects randomly [52]. The quasi experiment presented in this work was conducted with two groups of second-year computer science engineering students enrolled at the University of Bari during the first semester of the academic year 2018-2019. This section reports the experiment definition, design and settings in a structured fashion, following the well-known templates and guidelines provided in [52]–[54].

A. GOAL

The purpose is evaluating if the proposed approach supports source code development effectively and allows students to improve the correctness of the source code produced in performing development activity. To this aim, we investigate the differences between the performances of students supported by SPT tool (hereinafter referred to as students supported by SPT) and students that do not use the SPT tool and were supported, as usual, by teachers via email or in live meetings during office hours (hereinafter referred to as students not supported by SPT).



State and Behaviour Design – Missed Constant

Explanation
- The attribute *numMaxCopie* is never read or accessed by any method: use the static modifier, in combination with the final modifier, to define *numMaxCopie* as a constant. The final modifier indicates that the value of this field cannot change.

Abstraction and Hierarchy – Missed Generalization Relationship

Explanation
- Encapsulation: The fields *nome*, *cognome*, *annoNascita* and *LuogoNascita* are in common between Cliente and Editore classes. It can possible to define a superclass defining and implementing behavior common to Cliente and Editore classes.
- Polymorphism: Using Inheritance permits to use Polymorphism.
- Reusability: Using Inheritance helps in reusing code.
- Efficiency: It is efficient to use Inheritance while writing code. This can increase the speed of project.

Typing – Missed Inclusion Polymorphism

Explanation
- Polymorphism: Simultaneous presence of an *if-else-if* ladder and that of the *instanceof* in the source code { …} probably means you have failed to apply polymorphic behavior effectively. Using polymorphsim in the presence of inheritance make large case statements unnecessary, because each object implicitly knows its own type;
- Late binding: The ability to invoke methods depends only on the class from which an object was created, and not on the variable that stores a reference to it.

Encapsulation - Wrong Visibility of Class Member

Explanation
- Coupling: Use the most restrictive access level that makes sense for *rivista*, *cliente* and *mesePrestito* members. Use private unless you have a good reason not to;
- Flexibility: Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Bad Code Smells – Unnecessary Object Casting

Explanation
- In the following code *Persona pers = (Persona) new Cliente();* you used typecasting explicitly when it is not needed at all.

FIGURE 5. Examples of several process-level reports grouped by broad categories.

The quality focus regards how feedback provided by SPT affects the capability of developers to correctly apply the Object-Oriented constructs in the source code. We used the Goal-Question-Metric template [55], to define experimental goals (the object of study, purpose, focus, and the viewpoint of the measurement). Following this template, the goal is defined as follows:

"Analyze the source code produced by students supported by SPT and the one produced by students not supported by SPT to evaluate its correctness about the performance of the students in a development task in the context of a Programming II course held in the Computer Science Engineering first-level degree at University of Bari."

B. CONTEXT

The Programming II course is mandatory for all second-year Computer Science Engineering students of the University of Bari enrolled in Computer Science first-level degree. It is important to mention that the students are required to pass Programming I, Computer Architecture and Operating Systems courses to be enrolled in this course. Those requirements ensure that all the students have a similar academic background. Furthermore, firstly we asked students to answer a profiling questionnaire to exclude students that had previous experience in the Object-Oriented paradigm, and secondly, since students could have different programming skills and abilities, they were asked to participate in a preliminary training session to assess their entry-level in the course. During this training, session subjects were asked to complete a development task similar to those assigned during the empirical investigation. Since we are interested in studying the effect on development in terms of correctness and source code quality after training subjects with the proposed environment, using both results of profiling questionnaire and training session we selected only students that had no previous experience in Object-Oriented paradigm and able to understand the assignment, use the IDE and complete at least a 20% of the development task. Indeed, we assumed that students that already know the Object-Oriented paradigm could represent a too wide source of variability for a controlled in-lab experiment, as different starting levels could be influenced differently. Instead, we wanted all the subjects to start from the same level and work under the same initial conditions. Specifically, 87 students were enrolled in the Programming II course held during the first semester of 2018. According to the answers given to the profiling questionnaire and the results of the initial training session we selected all the students included two categories:

- **High ability subjects (HAS)**: those who had an academic score of at least 25/30 and completed correctly at least 50% of the development task;
- **Low ability subjects (LAS)**: the subjects that completed correctly at least the compulsory portion of the assignment (25% of the development task).

The rest of the subjects were excluded.

After all the tasks were completed, we selected 20 students having the following characteristics:

- all of them participated to the profile questionnaire;
- 10 of them are classified as HAS basing on the results of the profile questionnaire;
- 10 of them are classified as LAS basing on the results of the profile questionnaire.

The 20 students are then assigned to two equal groups of 10 persons, identified as 'Group A' and 'Group B'. Each group is composed of 5 students classified as LAS and other 5 students classified as HAS in profile questionnaire. This randomization process allows to make well-balanced groups: each group contains 10 students that are always made up of five "high ability" students and five "low ability" students.

All the students included in "Group A" received a customized feedback report produced by SPT on their outcomes, after each task. Differently, all the students of "Group B", never received feedback by SPT but only the feedback provided by teachers.

The experimental objects were the assignments that the students have to take when enrolled in the programming course. The assignments were designed by the teaching staff of the programming course and each student had to take six assignments: each assignment consisted of a development task to be performed from scratch in 120 minutes.

C. HYPOTHESES FORMULATION

By the study definition reported above, we formulated the following research hypotheses:

- H_0: there is no significant difference between the correctness of developed source code by students of Group A and the correctness of developed source code by students of Group B;
- H_1: $\neg H_0$

D. VARIABLE SELECTION

In this study, there is a single dependent variable: the source code correctness. The experiment prescribes, in each of several tasks, to develop from scratch a little software system written in Java using object-oriented constructs. To measure source code correctness we defined a metric model, as explained in the next section.

E. MEASUREMENT MODEL

In Table 1 we used a checkmark both to indicate the metrics whose measures were normalized (column titled *normalized*) and the metrics collected during our empirical investigation (column titled *experimentation*). The values of measures *ma*, *mo*, *mac*, *mgr*, *mar*, *mip*, *wubtp*, *wvcm* and *wubtp* have been normalized to be compared. For this reason, we calculated the aforementioned metrics as a complementary percentage, a particular percentage useful for finding out a missing percentage. Generally speaking, a percentage of occurrences of an element requires, to be calculated, to know the maximum numbers of occurrences of that element. For this reason, instructors, before the experiment sessions, carefully designed and implemented their "reference solution" for each assigned task to know the total number of mandatory properties and constraints that the solution must satisfy (in terms of relationships, attributes, operations, kind of elements, and so on). The metrics in the set *{oc,mc,pi,dp,cwif,upf}*, instead, are not normalized due to their intrinsic meaning. For instance, let us consider the *oc* metric: in a correct solution, there should be zero unnecessary object casting.

F. EXPERIMENTAL DESIGN

The proposed quasi-experiment is a between-subjects design: this choice is to minimize the learning and transfer across conditions (keeping control and treatment groups separate). The choice of between-subject design also implies that we consider two different groups of subjects, each of which performs similar tasks using only one technique (i.e., supported by SPT or not supported by SPT). This also forces to make sure that participants are allotted randomly to conditions, to ensure that subjects assignment does not affect study results (as already specified in this section, random selection was performed basing on ability). The structure of the quasi-experiment is depicted in Table 2. The experiment is made up of a sequence of six development sessions (pre-test, Lab 1-4, post-test). In the training test session, also called pre-test session, all the students used Eclipse and no one received feedback report at the end of the session. In the other sessions, the students of *GroupA* were supported by feedback provided by SPT at the end of the session while the students of *GroupB* never received any feedback by SPT. Different software systems specifications (S1,..., S6) are assigned during the sessions ensuring that each group performs developing one time on each system (the mapping among groups, systems, and sessions is reported in the Table rows).

TABLE 2. Experimental design of the quasi-experiment.

	Group A (*10 subjects*) 5 HAS + 5 LAS	**Group B** (*10 subjects*) 5 HAS + 5 LAS
Mark 1 (pre-test)	S1/NOSPT	S1/NOSPT
Lab 1	S2/SPT	S2/NOSPT
Lab 2	S3/SPT	S3/NOSPT
Lab 3	S4/SPT	S4/NOSPT
Lab 4	S5/SPT	S5/NOSPT
Mark 2 (post-test)	S6/NOSPT	S6/NOSPT

Figure 6 reports the overall structure of this investigation. When the pre-test session starts, students never received feedback from SPT.

G. EXPERIMENTAL PROCEDURE AND MATERIAL

Before the experiment, we asked the subjects to fill a profiling questionnaire in which we collect information about their ability and experience in programming, knowledge, and experience in object-oriented programming. Subjects have also participated in a training laboratory where they were asked to cope with a small development task very similar to the experiment's tasks. This made us confident that subjects were quite familiar with the development environment. The correctness in the training task has been recorded and used to assess the subjects' level of ability. To perform the experiment, subjects used a personal computer with a proxied connection and a browser capable to access the

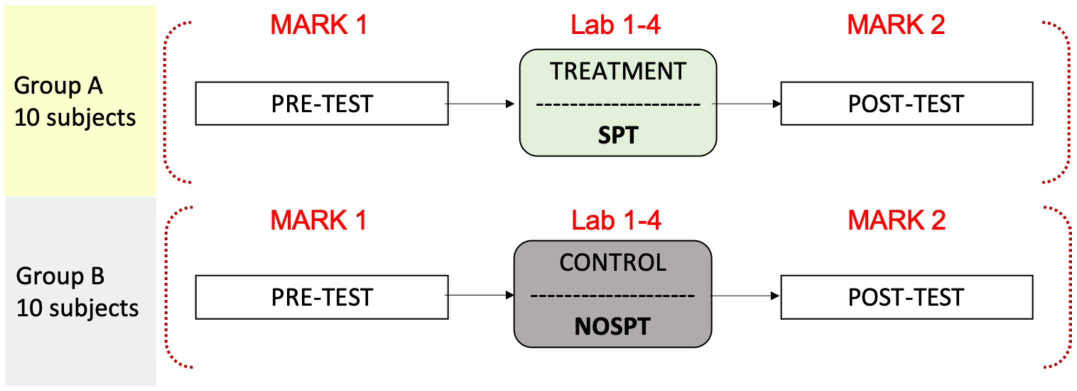


FIGURE 6. The overall structure of the between-subjects design.

SPT workspace front-end. Subjects were provided with the following material:

- a set of slides describing the experimental procedure;
- a document with the software requirements specification to be implemented.

Before the experiment, we gave subjects a clear description of the process to follow, but no reference was made to the study hypothesis. The experiment has been carried out according to the following procedure (executed by each student involved in the labs):

1) Read the document containing the software requirements to be implemented;
2) Mark the start time;
3) Develop the small system;
4) Mark the stop time;
5) Use SPT to create signed archives, in the projects data store, containing the produced source code and the development logs.

During the experiment, teaching assistants were present in the laboratory to prevent collaboration among subjects and to verify that the experimental procedure was fully respected.

H. ANALYSIS METHOD

We used a non-parametric statistical test to reject the null hypothesis H_0 related to the correctness of source code developed by subjects in performing development tasks. The use of a non-parametric test does not require that the population is normally distributed. Since the population is not paired (different subjects attended different labs) and distribution is not normally distributed, we used the Mann-Whitney U two-tailed test to check the hypotheses. Such a test allows checking whether differences exhibited by the two groups of subjects with different treatments (with SPT and without SPT) over the six sessions are significant. We assume significance at a 95% confidence level (=0.05), so we reject the null-hypothesis when $p\text{-}value < 0.05$.

To measure the magnitude of the treatment effect we measured the effect size. Statistics of effect size for the Mann-Whitney U-test report the degree to which one group has data with higher ranks than the other group. They are related to the probability that a value from one group will be greater than a value from the other group. To measure effect size we used Cliff's delta, whose values range from -1 to 1, with 0 indicating stochastic equality of the two groups. The extreme values (i.e., -1 and 1) indicates that one group shows complete stochastic domination over the other group. Finally, because multiple tests were performed, we adjusted the p-value with Holm-Bonferroni's correction [56].

VI. RESULTS

This section presents the experimental results.

The descriptive statistics related to Mark 2 reveal that correctness of source code developed by $Group_A$ was higher than for the $Group_B$. This can be seen in Figure 7. The box-plots show the students' performance in terms of every single metric defined to evaluate the correctness of the development tasks. In all cases, concerning Mark 1, there is an improvement in the learning performance (better scores). However, as can be observed, the improvement is much more consistent for the $Group_A$. Further considerations on the box-plots can be made:

- *mac, mgr and mar violations.* Concerning the fundamental relationships between these classes, the behavior of the two groups evolves in asymmetrically opposite way: the students of group A (those supported by the tool), understand the violations committed and, along with the experimentations, learn to apply them; the other students, those of group B, instead, do not show any improvement. This is confirmed by the violation distribution ranges that are, at Mark 2, consistently lower for group A (i.e., the group under treatment) than the group B.
- *cwif, mc, dp, wvcm.* Basic declarations such as instance fields declaration (cwif), constants declaration (mc), packages declaration (dp) and visibility of class members declaration (wvcm) show two different distributions: all students of Group A learned to well apply them (as shown by their lower distribution of violation count at Mark 2). Conversely, students of Group B, while still improving their skills, exhibit worse performances than the treatment group.

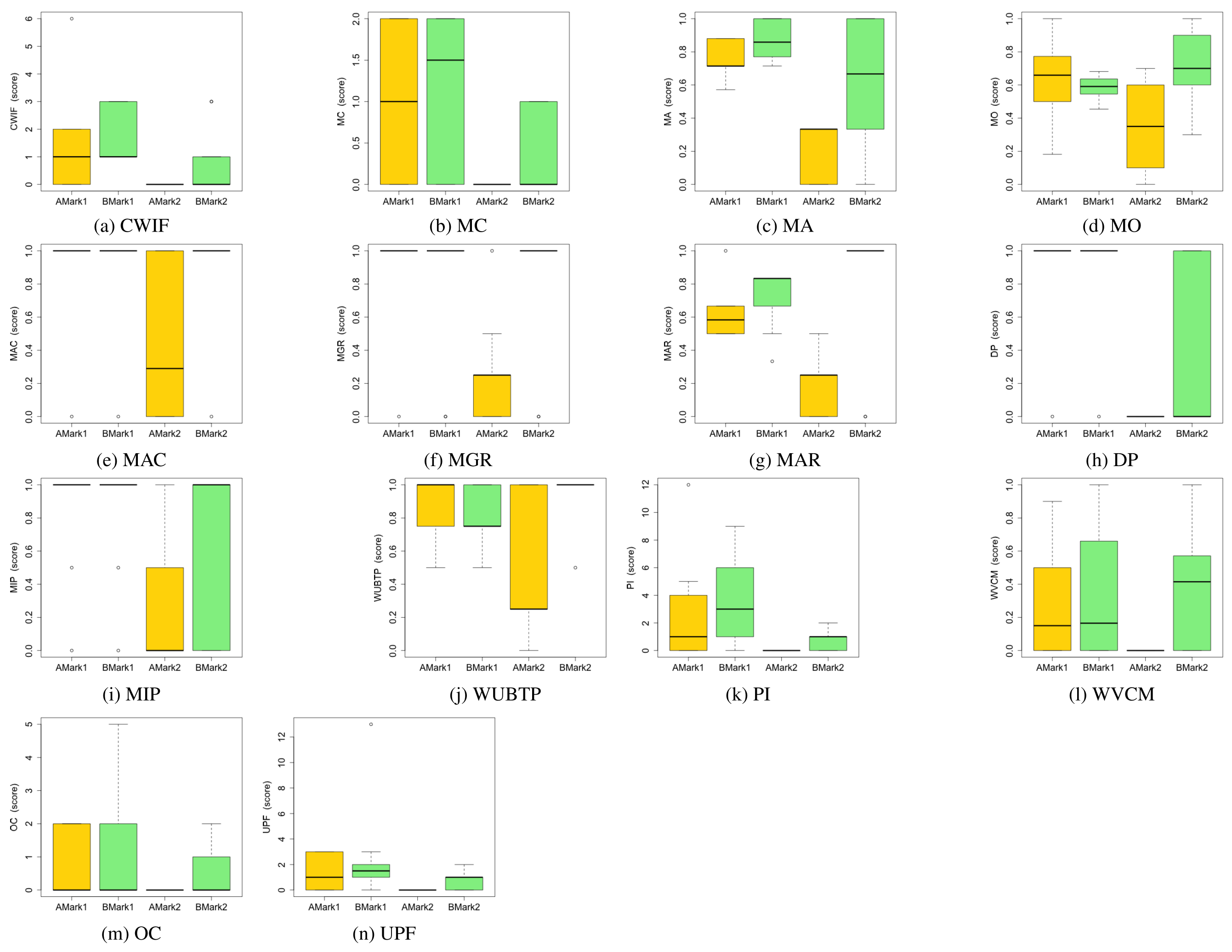


FIGURE 7. The box-plots of the results obtained in Mark 1 and in Mark 2 for language foundation metrics.

TABLE 3. T-test for correctness: $Group_A$ vs $Group_B$ (first part) - green dots indicate a statistically significant difference.

	cwif		mc		ma		dp	
Comparisons	*P*	*D*	*P*	*D*	*P*	*D*	*P*	*D*
$Group_A$/Mark1 vs $Group_B$/Mark1	0,43🔴	-0,32	0,46🔴	-0,19	0,07🔴	-0,48	1🔴	0
$Group_B$/Mark2 vs $Group_B$/Mark1	0,03🟢	-0,54	0,04🟢	-0,5	0,00 🟢	-1	0,02🟢	-0,5
$Group_A$/Mark2 vs $Group_A$/Mark1	0,00🟢	-0,64	0,00🟢	-0,60	0,00 🟢	-1	0,00🟢	-0,8
$Group_A$/Mark2 vs $Group_B$/Mark2	0,03🟢	-0,4	0,03🟢	-0,40	0,00🟢	-0,69	0,03🟢	-0,4

- *oc, upf*. The object casting and the use of private fields, the only ones bad smell-code violations measured in the experimentation, result to be the easiest concepts to apply for novice programmers. The two box-plots at Mark 1 are the lowest ones with the lower maxim values. At Mark 2, these violations completely disappear for group A while persist in a lower measure for group B.

The results of the t-test are shown in Tables 3 - 6 where the tests with a statistically significant difference between the two groups are highlighted with a green dot. The rows of these tables show the results of the t-test executed at Mark 1 and Mark 2 (comparing all metrics) for groups $Group_A$ and $Group_B$. We compared the two groups across different marks to:

- reject the null hypothesis that states that the course did not affect the correctness performances (second and third rows in the Tables).

We also compared the two groups on the same marks (first and fourth rows in the aforementioned Tables) to:

- ensure that the subjects were correctly balanced and did not start with a significant difference in performances (Mark 1);
- investigate if SPT treatment was more effective than classical training.

TABLE 4. T-test for correctness: *$Group_A$* vs *$Group_B$* (second part) - green dots indicate a statistically significant difference.

Comparisons	mo		mac		mgr		mar	
	P	*D*	*P*	*D*	*P*	*D*	*P*	*D*
$Group_A$/Mark1 vs $Group_B$/Mark1	0,42🔴	0,22	1🔴	0	0,30🔴	0,2	0,08🔴	-0,45
$Group_B$/Mark2 vs $Group_B$/Mark1	0,04🟢	-0,53	0,01🟢	-0,56	0,10🔴	-0,42	0,00🟢	-0,97
$Group_A$/Mark2 vs $Group_A$/Mark1	0,03🟢	-0,57	0,01🟢	-0,56	0,00🟢	-0,74	0,00🟢	-0,95
$Group_A$/Mark2 vs $Group_B$/Mark2	0,00🟢	-0,72	0,01🟢	-0,56	0,02🟢	-0,58	0,00🟢	-0,68

TABLE 5. T-test for correctness: *$Group_A$* vs *$Group_B$* (third part) - green dots indicate a statistically significant difference.

Comparisons	mip		wubtp		pi		wvcm	
	P	*D*	*P*	*D*	*P*	*D*	*P*	*D*
$Group_A$/Mark1 vs $Group_B$/Mark1	1🔴	0	0,47🔴	0,18	0,15🔴	-0,38	0,77🔴	-0,08
$Group_B$/Mark2 vs $Group_B$/Mark1	0,04🟢	-0,46	0,04🟢	-0,51	0,02🟢	-0,58	0,04🟢	-0,45
$Group_A$/Mark2 vs $Group_A$/Mark1	0,00🟢	-0,73	0,02🟢	-0,57	0,00🟢	-0,60	0,01🟢	-0,50
$Group_A$/Mark2 vs $Group_B$/Mark2	0,03🟢	-0,51	0,00🟢	-0,66	0,00🟢	-0,70	0,00🟢	-0,60

TABLE 6. T-test for quality: *$Group_A$* vs *$Group_B$* - green dots indicate a statistically significant difference.

Comparisons	upf		oc	
	P	*D*	*P*	*D*
$Group_A$/Mark1 vs $Group_B$/Mark1	0,32🔴	-0,36	0.79🔴	-0.07
$Group_B$/Mark2 vs $Group_B$/Mark1	0,03🟢	-0,53	0,69🔴	-0,1
$Group_A$/Mark2 vs $Group_A$/Mark1	0,02🟢	-0,53	0,03🟢	-0,4
$Group_A$/Mark2 vs $Group_B$/Mark2	0,02🟢	-0,51	0,03🟢	-0,4

For Tables 3 - 6, we used *P* to indicate the *p-value adjusted with Holm-Bonferroni's correction* [56] and *D* to indicate the effect size evaluated using *Cliff's Delta*.

Observing the Tables the following considerations can be made:

- the null hypothesis H_0 can be rejected in all cases; in particular with a level of significance below the alpha threshold set at 0.05 for the metrics *cwif*, *mc*, *upf*, *mac*, *mgr*, *mip*, *wvcm*, *dp* and *oc* and with a level of significance below the alpha threshold set at 0.01 for all the remaining metrics;
- before the course there was no statistically significant difference between the two groups' performances because the level of significance is below 95% for $Group_A$ at Mark 1 and $Group_B$ at Mark 1;
- the course only in two cases (*mgr* and *oc*) does not have a significant effect on source code correctness and only for students of $Group_B$. In all remaining cases at all marks, the level of significance of the test is over 95%.

VII. DISCUSSION OF THE RESULTS AND INTERPRETATION OF FINDINGS

In this section, the results are discussed and the interpretations of findings are provided.

Concerning the hypothesis H_0 and H_1 the data show that the student's outcomes always improve in terms of correctness and quality, except for *mgr* and *oc* violations related to $Group_B$. Indeed, for all the violations measured in the empirical investigation, there is a statistically significant difference, as shown in tables 3-6, between students who received feedback by SPT and those who did not. Figure 7 shows how source code developed by students supported by SPT always improves in terms of correctness more than students that were trained not using SPT. The results show that the SPT is useful for highlighting the violations committed by students in their OO programs instructing them about what they did wrong and how to improve. The results confirm that students who have been exposed only to procedural programming have considerable difficulty in understanding the basic concepts of OO. These difficulties range from class design to more advanced object-oriented concepts such as hierarchy and typing. The results obtained in Mark 2 show that violations related to misused foundation constructs of the language are completely absent for students of the $Group_A$, unlike those of $Group_B$. These violations are *mc*, *upf*, *wvcm*, *pi*, *dp*, *cwif* and *oc*.

For all the remaining violations, student's improvements are still statistically significant, but being more related to high-level constructs requiring more experience, they continue to occur. This may be since understanding and not committing these violations require more effort (in terms of both comprehension of the concepts and ability to apply them effectively).

The analysis of results drives some further considerations, grouped by broad categories of violations:

- **State and Behaviour Design**: results highlight that state management issues are addressed much more effectively than behavioral ones by a report-driven approach like the one implemented in SPT. Reports provided by SPT suggest, for instance, students to rethink how they declare a variable in Java and solicit them to avoid the usage of ordinary fields that hide a "constant" semantics. An example is shown in Figure 5 where the report excerpt for *State and Behaviour Design - Missed Constant* violation highlights that "the attribute numMaxCopie is never read or accessed by any method" and suggests the use of the static modifier.
- **Abstraction and Hierarchy**: students and novice developers, often misuses generalization. Our results, on this

point, confirm the findings in [57]. The reports highlight the advantages of using generalization, as shown in Figure 5 for *Missed Generalization Relationship*, in terms of encapsulation, reusability and efficiency. This probably has motivated students to think about using Abstraction and Hierarchy. The effect of reporting such issues helps students, like Figure 7 shows, to drastically reduce the number of errors.

- **Typing**: novices often use explicit type inference that could be avoided. This is a sign that the students have failed to apply polymorphic behavior effectively. It is interesting to note that this violation category, even if improved in a significant way using SPT reports, it is still present in the produced source code even at Mark 2. This suggests improving reports automatically generating counter-examples to students on their source-code also highlighting the negative consequences of their solutions. The feedback provided highlights how using polymorphism in the presence of inheritance helps to reduce code produced and to make it more readable as suggested in the example of Figure 5 for *Typing - Missed Inclusion Polymorphism*.
- **Encapsulation**: the source code developed by the students shows that the students in many cases do not explicitly use access level modifiers for class members, probably because they do not have previous experience related to the access levels of an element especially for large software systems. Reports include information about coupling and highlight scalability problems to make them aware of this kind of mistakes as shown in Figure 5 for *Encapsulation - Wrong Visibility Class Member* violation.
- **Bad Code Smells**: Reports suggest to avoid writing unnecessary code as stated in the example shown in Figure 5 for *Bad Code Smells - Unnecessary Object Casting*.

VIII. THREATS TO VALIDITY

To judge the quality of our work it is necessary to consider the following threats to the validity of the study:

- **External validity**. In our case, global generalization is not possible, particularly as human beings have taken part in the experimentation. The results could, therefore, be valid in similar contexts, such as Java programming courses during first-year of computer science degrees. Concerning the representativeness of the materials used, all the experimental materials were academic exams. The experience and professionalism of the teaching staff involved have, in our opinion, alleviated any possible lack of quality or objectivity when creating these materials. We are aware that different assignments might have different complexity but, as previously stated, the lecturer in charge of the final review of the exams is one of the authors of this work and particular attention was, therefore, paid to creating exams with a similar level of complexity. The generalization to other populations of students (e.g., cohorts with different levels of motivation, or different educational backgrounds, different programming languages, etc.), is worthy of future experimentation. Finally, if the reference solution is poorly defined the approach performs badly since is following wrong directives.
- **Internal validity**. One possible threat is that the subjects were assigned to one group or another depending on the academic year in which they took the programming course. Our experience as lecturers indicates that there is normally no significant difference between the backgrounds of two consecutive academic years. Another possible threat is that the subjects could have attended programming classes at secondary school, signifying that the results could have been threatened by these subjects. However, this threat appears to be just as probable for all groups of subjects and we, therefore, assume that there should be no difference in the results obtained in the reported experiment. Moreover, subjects were not aware of the experimental hypotheses and were not rewarded for the participation in the experiment. Finally, the small sample size we used (ten subjects for each group) represents a threat because it could reduce the power of the study and increase the margin of error, which can render the study meaningless.
- **Construct validity**. The dependent variable (i.e., *the correctness of source code developed*), might have been affected by the teaching staff on the programming courses and the pedagogical methods used by these professors. To avoid this possible threat, the teaching staff were the same people throughout the data collection period, signifying that the pedagogical methods used by each professor will probably have affected the learning effectiveness in both cases (supported and not supported by SPT). Nevertheless, to mitigate this threat, all the exams were created by a researcher who has coordinated all the programming teaching staff for more than 10 years; this individual is also one of the authors of the present work. We are also aware that support provided by SPT can be more motivating for students, which might lead to more time spent practicing, doing exercises, and so on, and thus a better grade in exams. To mitigate this issue, we decided to limit the support of SPT only to planned sessions in which all the students were involved. Finally, it is important to highlight that the metric model used does not cover all defined violations since many other violations could be defined and measured.
- **Conclusion validity**. The data were collected by two researchers, co-authors of this work, which supervised all the development sessions performed by the students. Statistical tests were used to reject the null hypotheses, fulfilling all the requirements needed to do so, signifying that the validity of the results obtained is acceptable. About the statistical power required to accept these results, the number of experimental

subjects was sufficiently large for us to achieve an acceptable statistical power in these tests. Specifically, we used non-parametric statistical tests (Mann-Whitney and Wilcoxon) that do not require normal distribution of the experimental data and we also applied the Holm-Bonferroni [56] correction to counteract the problem of multiple comparisons.

IX. CONCLUSION

We have proposed an approach to analyze the student's source code, identify the student's mistakes and misconceptions, support students by feedback explaining why they committed mistakes and how to address them.

The contribution of this article is also to present an experiment that was undertaken to gather empirical evidence on the beneficial effects of the proposed approach on the learning process on an OOP course in the second-year of the computer science engineering degrees at the University of Bari. The results show that the SPT can support the learning process increasing its effectiveness. In particular, SPT gives encouraging results as regards the correctness of the code produced by the students since the students using the tool obtained higher grades than those using traditional teaching. Despite the significant results obtained in this experiment, we are aware that replication is useful to corroborate and strengthen our findings.

REFERENCES

[1] *Joint Task Force on Computing Curricula ACM IEEE Computer Society, Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://dl.acm.org/doi/book/10.1145/2534860

[2] H. Roumani, "Practice what you preach: Full separation of concerns in CS1/CS2," in *Proc. 37th Tech. Symp. Comput. Sci. Edu. (SIGCSE)*, New York, NY, USA, 2006, pp. 491–494, doi: 10.1145/1121341.1121495.

[3] D. J. Armstrong, "The quarks of object-oriented development," *Commun. ACM*, vol. 49, no. 2, pp. 123–128, Feb. 2006.

[4] K. Sanders and L. Thomas, "Checklists for grading object-oriented CS1 programs: Concepts and misconceptions," in *Proc. 12th Annu. SIGCSE Conf. Innov. Technol. Comput. Sci. Edu. (ITiCSE)*, New York, NY, USA, 2007, pp. 166–170, doi: 10.1145/1268784.1268834.

[5] P. Ardimento, M. L. Bernardi, and M. Cimitile, "On the students' misconceptions in object-oriented language constructs," in *Proc. Higher Edu. Learn. Methodologies Technol. Online 1st Int. Workshop (HELMeTO)*, Novedrate, CO, Italy, vol. 1091. Cham, Switzerland: Springer, Jun. 2019, 2019, pp. 97–112, doi: 10.1007/978-3-030-31284-8_8.

[6] T. Clear, "Diagnosing your teaching style: How interactive are you?" *ACM Inroads*, vol. 1, no. 2, pp. 34–41, Jun. 2010.

[7] S. Beecham, J. Noll, and T. Clear, "Do we teach the right thing? A comparison of GSE education and practice," in *Proc. IEEE 12th Int. Conf. Global Softw. Eng. (ICGSE)*, May 2017, pp. 11–20.

[8] G. Wiggins, "Seven keys to effective feedback," *Educ. Leadership*, vol. 70, no. 1, pp. 11–16, 2012.

[9] A. Settle and J. Glatz, "Rethinking advising: Developing a proactive culture to improve retention," in *Proc. Conf. Inf. Technol. Edu. (SIGITE)*, 2011, pp. 9–14.

[10] N. C. C. Brown and A. Altadmri, "Novice java programming mistakes: large-scale data vs. Educator beliefs," *ACM Trans. Comput. Edu.*, vol. 17, no. 2, pp. 1–21, Jun. 2017.

[11] N. Ragonis and M. Ben-Ari, "A long-term investigation of the comprehension of OOP concepts by novices," *Comput. Sci. Edu.*, vol. 15, no. 3, pp. 203–221, Sep. 2005, doi: 10.1080/08993400500224310.

[12] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," in *Proc. 17th ACM Annu. Conf. Innov. Technol. Comput. Sci. Edu. (ITiCSE)*, 2012, pp. 75–80.

[13] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Trans. Comput. Edu. (TOCE)*, vol. 13, no. 4, p. 19, 2013.

[14] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing syntax error messages appears ineffectual," in *Proc. Conf. Innov. Technol. Comput. Sci. Edu. (ITiCSE)*, 2014, pp. 273–278.

[15] B. A. Becker, G. Glanville, R. Iwashima, C. McDonnell, K. Goslin, and C. Mooney, "Effective compiler error message enhancement for novice programming students," *Comput. Sci. Edu.*, vol. 26, nos. 2–3, pp. 148–175, Jul. 2016.

[16] B. A. Becker, C. Murray, T. Tao, C. Song, R. McCartney, and K. Sanders, "Fix the first, ignore the rest: Dealing with multiple compiler error messages," in *Proc. 49th ACM Tech. Symp. Comput. Sci. Edu.*, Feb. 2018, pp. 634–639.

[17] R. S. Pettit, J. Homer, and R. Gee, "Do enhanced compiler error messages help students?: Results Inconclusive.," in *Proc. ACM SIGCSE Tech. Symp. Comput. Sci. Edu.*, Mar. 2017, pp. 465–470.

[18] D. McCall and M. Kölling, "Meaningful categorisation of novice programmer errors," in *Proc. IEEE Frontiers Edu. Conf. (FIE)*, Oct. 2014, pp. 1–8.

[19] A. Ettles, A. Luxton-Reilly, and P. Denny, "Common logic errors made by novice programmers," in *Proc. 20th Australas. Comput. Edu. Conf. (ACE)*, New York, NY, USA, 2018, pp. 83–89, doi: 10.1145/3160489.3160493.

[20] J. Spacco, P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, and R. Duvall, "Analyzing student work patterns using programming exercise data," in *Proc. 46th ACM Tech. Symp. Comput. Sci. Edu. (SIGCSE)*, 2015, pp. 18–23.

[21] J. Bulmer, A. Pinchbeck, and B. Hui, "Visualizing code patterns in novice programmers," in *Proc. 23rd Western Can. Conf. Comput. Edu. (WCCCE)*, New York, NY, USA, 2018, pp. 1–6, doi: 10.1145/3209635.3209652.

[22] P. Ardimento, M. L. Bernardi, M. Cimitile, and F. M. Maggi, "Evaluating coding behavior in software development processes: A process mining approach," in *Proc. IEEE/ACM Int. Conf. Softw. Syst. Processes (ICSSP)*, May 2019, pp. 84–93, doi: 10.1109/ICSSP.2019.00020.

[23] P. Ardimento, M. L. Bernardi, M. Cimitile, and G. De Ruvo, "Mining developer's behavior from Web-based IDE logs," in *Proc. IEEE 28th Int. Conf. Enabling Technol. Infrastruct. Collaborative Enterprises (WETICE)*, Jun. 2019, pp. 277–282, doi: 10.1109/WETICE.2019.00065.

[24] N. Ragonis and M. Ben-Ari, "On understanding the statics and dynamics of object-oriented programs," in *Proc. ACM SIGCSE Bull.*, vol. 37, 2005, pp. 226–230.

[25] N. Liberman, C. Beeri, and Y. B.-D. Kolikant, "Difficulties in learning inheritance and polymorphism," *ACM Trans. Comput. Edu.*, vol. 11, no. 1, pp. 4:1–4:23, Feb. 2011, doi: 10.1145/1921607.1921611.

[26] M. Madden and D. Chambers, "Evaluation of student attitudes to learning the java language," in *Proc. Inaugural Conf. Princ. Pract. Program., 2nd Workshop Intermediate Represent. Eng. Virtual Mach. (PPPJ)*, 2002, pp. 125–130. [Online]. Available: http://dl.acm.org/citation.cfm?id=638476.638501

[27] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proc. ACM Conf. Innov. Technol. Comput. Sci. Educ.*, 2017, pp. 110–115.

[28] S. H. Edwards, N. Kandru, and M. B. M. Rajagopal, "Investigating static analysis errors in student java programs," in *Proc. ACM Conf. Int. Comput. Edu. Res. (ICER)*, New York, NY, USA, Aug. 2017, pp. 65–73, doi: 10.1145/3105726.3106182.

[29] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students," *SIGCSE Bull.*, vol. 35, no. 1, pp. 153–156, Jan. 2003. [Online]. Available: http://doi.acm.org/10.1145/792548.611956

[30] T. Buckers, C. Cao, M. Doesburg, B. Gong, S. Wang, M. Beller, and A. Zaidman, "UAV: Warnings from multiple automated static analysis tools at a glance," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2017, pp. 472–476.

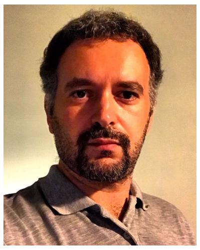
[31] J. Hattie and H. Timperley, "The power of feedback," *Rev. Educ. Res.*, vol. 77, no. 1, pp. 81–112, Mar. 2007, doi: 10.3102/003465430298487.

[32] C. Ott, A. Robins, and K. Shephard, "Translating principles of effective feedback for students into the CS1 context," *ACM Trans. Comput. Edu.*, vol. 16, no. 1, pp. 1–27, Feb. 2016, doi: 10.1145/2737596.

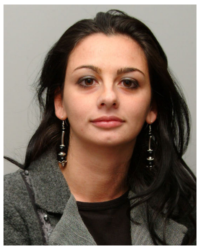
[33] D. Carless, D. Salter, M. Yang, and J. Lam, "Developing sustainable feedback practices," *Stud. Higher Edu.*, vol. 36, no. 4, pp. 395–407, Jun. 2011.

[34] W. K. Balzer and M. E. Doherty, "Effects of cognitive feedback on performance," *Psychol. Bull.*, vol. 106, no. 3, p. 410, 1989.
[35] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *Proc. 45th ACM Tech. Symp. Comput. Sci. Edu. (SIGCSE)*, New York, NY, USA, 2014, pp. 223–228, doi: 10.1145/2538862.2538924.
[36] N. C. C. Brown, A. Altadmri, S. Sentance, and M. Kölling, "Blackbox, five years on: An evaluation of a large-scale programming data collection project," in *Proc. ACM Conf. Int. Comput. Edu. Res.*, Espoo, Finland, Aug. 2018, pp. 196–204.
[37] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston, *Object-Oriented Analysis and Design With Applications*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2007.
[38] C.-L. Chen, S.-Y. Cheng, and J. M.-C. Lin, "A study of misconceptions and missing conceptions of novice java programmers," in *Proc. Int. Conf. Frontiers Educ. Comput. Sci. Comput. Eng. (FECS)*, 2012, p. 1.
[39] N. Ragonis and M. Ben-Ari, "Teaching constructors: A difficult multiple choice," in *Proc. 16th Eur. Conf. Object-Oriented Program., Workshop*, vol. 3, 2002, pp. 152–176.
[40] S. Xinogalos, "Object-oriented design and programming: An investigation of novices' conceptions on objects and classes," *ACM Trans. Comput. Edu.*, vol. 15, no. 3, p. 13, 2015.
[41] E. Tempero, S. Counsell, and J. Noble, "An empirical study of overriding in open source java," in *Proc. 33rd Australas. Conf. Comput. Sci.*, vol. 102, 2010, pp. 3–12.
[42] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," *Comput. Sci. Edu.*, vol. 15, no. 2, pp. 83–102, Jun. 2005.
[43] M. Snoeck and G. Dedene, "Generalization/specialization and role in object oriented conceptual modeling," *Data Knowl. Eng.*, vol. 19, no. 2, pp. 171–195, 1996, doi: 10.1016/0169-023X(95)00044-S.
[44] P. A. Sivilotti and M. Lang, "Interfaces first (and foremost) with java," in *Proc. 41st ACM Tech. Symp. Comput. Sci. Edu.*, 2010, pp. 515–519.
[45] M. Fowler. (2006). *Code Smell*. Accessed: Jun. 28, 2020. [Online]. Available: https://martinfowler.com/bliki/CodeSmell.html
[46] R. Biddle and E. Tempero, "Java pitfalls for beginners," *ACM SIGCSE Bull.*, vol. 30, no. 2, pp. 48–52, Jun. 1998.
[47] N. Ragonis and R. Shmallo, "On the (MIS) understanding of the this reference," in *Proc. ACM SIGCSE Tech. Symp. Comput. Sci. Educ.*, 2017, pp. 489–494.
[48] J. Bergin, A. Agarwal, and K. Agarwal, "Some deficiencies of C++ in teaching CS1 and CS2," *ACM SIGPLAN Notices*, vol. 38, no. 6, pp. 9–13, Jun. 2003.
[49] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *Proc. 12th Eur. Conf. Softw. Maintenance Reeng.*, Apr. 2008, pp. 329–331.
[50] CSELAB. 2018. *Student Profiling Tool*. [Online]. Available: https://gitlab.com/cselab/spt
[51] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Trans. Comput. Edu.*, vol. 19, no. 1, pp. 3:1–3:43, Jan. 2019, doi: 10.1145/3231711.
[52] T. D. Cook and D. T. Campbell, "The causal assumptions of quasi-experimental practice," *Synthese*, vol. 68, no. 1, pp. 141–180, Jul. 1986. [Online]. Available: https://doi.org/10.1007/BF00413970
[53] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 201–228.
[54] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Cham, Switzerland: Springer, 2012.
[55] V. R. Basili, G. Caldiera, and D. H. Rombach, *The Goal Question Metric Approach*, vol. 1. Hoboken, NJ, USA: Wiley, 1994, ch. 1, pp. 234–263.
[56] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandin. J. Statist.*, vol. 6, no. 2, pp. 65–70, 1979. [Online]. Available: http://www.jstor.org/stable/4615733
[57] J. Bennedsen and M. E. Caspersen, "Abstraction ability as an indicator of success for learning object-oriented programming?" *SIGCSE Bull.*, vol. 38, no. 2, pp. 39–43, Jun. 2006, doi: 10.1145/1138403.1138430.

PASQUALE ARDIMENTO was born in Bari, Italy. He received the Ph.D. degree in computer science from the University of Bari. In 2004, he collaborated as a Research Student with the Department of Engineering, University of Durham, U.K. From 2005 to 2007, he was a Contract Researcher with the University of Bari. Since 2008, he has been a Researcher with the Computer Science Department, University of Bari Aldo Moro. He has authored more than 60 articles published in journals and conference proceedings. His main research interests include software engineering (maintenance, testing, data mining on software systems, software quality assurance, and computational intelligence). He serves both as a member of the program and organizing committees of conferences and as a Reviewer for articles submitted to some of the main journals and magazines in the field of software engineering and software maintenance. He was involved in several national research projects and European projects and served as a Reviewer for many international and national conferences. He is a member of the Consorzio Interuniversitario Nazionale per l'Informatica (CINI).



MARIO LUCA BERNARDI (Member, IEEE) received the Laurea degree in computer science engineering from the University of Naples "Federico II," Italy, in 2003, and the Ph.D. degree in information engineering from the University of Sannio, in 2007. Since 2003, he has been a Researcher in software engineering with University of Sannio. He has authored more than 70 articles published in journals and conference proceedings. His main research interests include software maintenance and testing, software development, and architectural design, with a particular interest in data mining, artificial intelligence, and computational intelligence. He has served and is serving in the program and organizing committees of conferences and as a Referee for main journals and magazines and has been involved in several projects on tasks related to software engineering, software maintenance, quality assurance, and artificial intelligence applied in different contexts and domains for both the industry and services sectors.



MARTA CIMITILE (Member, IEEE) received the Ph.D. degree in computer science from the Department of Computer Science, University of Bari, in 2008. She is currently an Assistant Professor and an Aggregated Professor with Unitelma Sapienza University, Rome, Italy. She received the Italian Scientific Qualification for the associate professor position in computer science engineering, in April 2017. In the last year, she was involved in several industrial and research projects. She is a Founding Member of the spin-off of the University of Bari named Software Engineering Research and Practices s.r.l. She published more than 60 articles at international conferences and journals. Her main research interests include business process management and modeling, knowledge modeling and discovering, and process and data mining in software engineering environment. She served in the program and organizing committees of several international conferences. She is a Reviewer for some of the main journals and magazines in the field of knowledge management and software engineering, knowledge representation and transfer, and data mining. She is on the Editorial Board of the *Journal of Information and Knowledge Management* and *PeerJ Computer Science*. She is an Associate Editor of IEEE ACCESS.