On the Students' Misconceptions in Object-Oriented Language Constructs

Pasquale Ardimento[1](✉), Mario Luca Bernardi[2], and Marta Cimitile[3]

[1] University of Bari Aldo Moro, Bari, Italy
pasquale.ardimento@uniba.it
[2] University Giustino Fortunato, Benevento, Italy
m.bernardi@unifortunato.eu
[3] Unitelma Sapienza, Rome, Italy
marta.cimitile@unitelmasapienza.it

Abstract. Analyze the Object-oriented (OO) source code developed by students provides useful formative tips to instructors. According to this, it is essential to understand the student's real difficulties allowing instructors to shape effective courses. To provide run-time feedback to students and to study and analyze the evolution of their performances offline and over time we designed a framework and developed a tool. It allows to identify students' misconceptions analysing source code and to create personalized student reports automatically. In this paper, we present an empirical study, conducted using our toolchain, that involves 1627 projects extracted from the multi-institution Blackbox dataset. We identified a violation model for Java language constructs based on established results in the computing education community. Afterwards, we grouped such violations in categories and analyzed the relations among them. Our contributions might be helpful in delivering formative feedback and supporting instructors who teach Java and object-oriented programming in general.

Keywords: Object-oriented · Misconceptions · Data analytics

1 Introduction

Since Java-based courses give the first experience in programming to many students, the teaching and knowledge transferring activity is thoroughly critical and arduous [2,10]. For this reason, there is a continuous research of novel teaching strategies allowing to address student questions as soon as possible providing timely feedback [32], and preventing retention. These strategies require a high comprehension of how student perform programming activities a their main difficulties.

Basing on the above considerations, we aim to understand how students use Java and its language constructs by analyzing the most common violations of the Object Oriented paradigm in student source code. Our study is in line with existing literature regarding the comprehension of student mistakes [4] and misconceptions [23] to shape the instructor's teaching strategy. The violations are identified from a sample of student source code and organized in categories covering the most relevant language constructs and tipical quality issues.

This paper proposes an empirical study to characterize the violations and to identify their relations. The empirical study addresses the following research questions:

RQ1: *What is the volume of violations of object-oriented language constructs in student source code?*
RQ2: *Are students who make mistakes in one category inclined to make mistakes in other categories as well?*

To perform the empirical study, we developed a static source code analysis framework and a supporting tool—called Student Profiling Tool (SPT)—to detect and report violations in student source code for both students (in real-time) and teachers (allowing offline automatic analyses).

To answer the RQs, we used SPT to analyze a sample of 1627 Java projects extracted from the Blackbox dataset [5]. The results of the analysis highlight interesting correlations among the violations belonging to different categories. We believe that these contributions might be useful to instructors, helping them to drive the development of novel teaching strategies improving the effectiveness of Object Oriented courses.

The rest of the paper is organized as follows. In the next section, we describe related work. In Sect. 3 we introduce the methodology for the creation of the list of violations and design and development of Student Profiling Framework, and Tool. Section 4 deals with the empirical study. Discussion and implications of our findings are in Sect. 5. Finally, Sect. 6 concludes the paper and suggests future work.

2 Related Work

Object-oriented source code errors have been widely investigated in the literature. Some studies are focused on the difficulties faced during the learning process of OOP [18].

In [12], the authors analyse 15,000 code fragments, generated by novice programming students. The logic errors have been classified as algorithmic errors, misinterpretations of the problem, and fundamental misconceptions.

Keuning et al. [17] examine the quality of student code with regards to program flow, functions, clarity of expressions, decomposition, and modularization. They state that novice programmers write source code characterized by significant quality issues and professional static analysis tools (Checkstyle[1], PMD[2], FindBugs[3], Sonar[4]), as currently designed, offer little or no help. Moreover, such tools appear to be troublesome for preparing new teaching strategies since they are considered confusing also for experts as to necessitate the development of an intuitive supporting environment to compare the results of their analyses [6].

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

Edwards et al. [11], consider the problems detected by several professional tools for analyzing the code of 3,691 students over five semesters.

Sanders and Thomas [25] propose two checklists for grading student programs. The checklists are obtained by considering basic object-oriented programming concepts and typical novice misconceptions as identified in the literature. The evaluation of the checklists is performed in an objects-first CS1 course.

Expresso is an error detection advisory tool [15]. It aims to help teachers to understand the types of frequent errors students make among a list of the typical logic, semantic and syntax errors usually made by novice programmers. Madden and Chambers [20] report a survey about the aspects of the Java language that students perceive to be most difficult. Conversely, in this paper, we present an empirical study to characterize Java language violations. In addition, our study focuses on the analysis of the relationships among categories of language constructs to understand the impact of misconceptions in one category concerning the others. Finally, we exploit a supporting tool that can produce a detailed report on the violations of the source code given as input.

3 Methodology

As we mentioned in the previous section, the literature widely addressed OO concepts and misconceptions of novice programmers. We started from such misconceptions of novice programmers to find how these cause mistakes and violations of language constructs. Similarly to [19,25], we manually inspected a sample of student programs, namely 162 (∼10%) of 1627 Java projects extracted from Blackbox [5]. We then turned to the literature highlighting, for each paper, what kind of misconceptions/errors to expect in student code, (e.g. instance/class conflation, problems with abstractions, issues with inheritance and polymorphism, difficulties with constructors, confusion using attributes and local variables, intricacies with scope). To facilitate our job, we extracted excerpts of code that could have helped us to focus more specifically on language constructs. Once we found good violation candidates, for a given language construct, we implemented a static analysis "recipe" to automatically look for further occurrences in arbitrary student source code.

Finally, after multiple iterations, we reached the list described in the following, and divided into categories.

3.1 Abstraction Violations

This category deals with wrong usage of abstraction. We considered the following violations:

- **Empty NOn-ABstract method in root class (enoab)** In Example 1.1, method `printSmth` is an empty non abstract method (*enoab*) and does not make any sense. This is not, in fact, the case "when classes provide empty implementations that override non-empty implementations." [29]. There is no overriding in the figure since `Enoab` is a root class.

- **Class With Implicit Constructor (cwic)** Ragonis and Ben-Ari [22] 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. Other simpler styles caused serious misconceptions.". This means that a class with an implicit constructor (*cwic*), listed among "simpler styles", should be avoided.
- **Class Without Instance Fields (cwif)** It is well-known that students have difficulties in understanding the concepts of object and class [23]. In particular, they have "difficulties in understanding the static aspect of the class definition". A class without instance fields (*cwif*) could be a consequence of such difficulties, like `class Cwif` in Example 1.2. Of course, a class without instance fields containing only a `main` method cannot be considered a violation.
- **Poor Interface usages and definitions (pi)** Among the "good coding practices for Java", Sivilotti et al. [27] 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). This is not the case of `b` in Example 1.3. Another poor use of interfaces is when there is only a single implementation [26].

Example 1.1. Empty NOn-ABstract method in root class

```
public class Enoab {

  public void printSmth(){}
  public int doSmth(){
        int easySum = 2 + 2;
        return easySum;
  }
}
```

Example 1.2. Class Without Instance Fields

```
public class Cwif {

  public void makeStuff(){ //...
  }
  public int makeOtherStuff(){
        return 0;
  }
}
```

Example 1.3. Poor Interfaces

```
public interface Bable { //...
}

// single implementation
public class B implements Bable {
     //...
}
public class B2 implements Bable
     { //...
}

public class A {
  public void doSmth() {
    // declared site should use an
         interface
         B b = new B();
  }
}
```

3.2 Attribute Violations

This category deals with wrong definition or use of fields. We considered the following violations:

- **Field Used as Local Variable (fulv)** Students have issues in understanding the "difference between class fields and local variables inside methods", as

stated by Biddle and Tempero [3]. A field written before being read (`fulv` in Example 1.4) is an effect of aforementioned issue. Such field should be indeed a local variable.
- **Missed Constant (mc)** A class field which is only read should be declared as constant—like `mc` in Example 1.5. This is in line with Chen et al. [7] who discovered misconceptions when students "determine which data member is appropriate for declaring as constant". Moreover, Ragonis and Ben-Ari [23] list "difficulties in understanding the static aspect of the class definition".
- **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*. This is the case of `shadFloat` in Example 1.6.
- **Public Field Changed by private methods (pfc)** "Difficulties understanding the influence of method execution on the object state" is another problem related to the concepts of object and class [23]. A consequence of aformentioned difficulties is `pfChanger()` in Example 1.7 where a private method changes a public field.
- **Unused Private Field (upf)** This is a well-know warning detected by popular IDEs but the fact that students still commit this violation means that instructors should focus more on this aspect. Reasons of this issue could be the same of *fulv*. Students maybe have still to figure out how to design a class and what should or should not be part of it.

Example 1.4. Field Used as Local Variable

```
public class ClassWithFULV {
  private int fulv = 1;
  public void
        methodUsingFULV(int
        c) {
         fulv = c + 3;
         if(fulv == 4) { //...
         }
  }
}
```

Example 1.5. Missed Constant

```
public class ClassWithMC {
  private String mc = "String";
  public method() {
    if (mc.equals("MISSED")){
          //...
    }
  } //other methods not writing
        on mc
}
```

Example 1.6. Local Variable Shadowing Field

```
public class LVSFClass {
  private float shadFloat = 0.0;

  public float methodF(){
        return shadFloat;
  }
  public void methodS(int d) {
//shadowing instance variable 'shadFloat' with a local variable with the
      same name
        float shadFloat = 0.5;
        float prod = 0.85*shadFloat;
  }
}
```

Example 1.7. Public Field Changed by private methods

```
public class ClassWithPFC {
   public int pfc = 1;
   private void pfChanger(){
             this.pfc = 0;
   }
}
```

Example 1.8. Inheritance to Extend Values

```
public class Bicycle {
   int wheelCount = 2;
   void gearDown(){ //...
   }
}
public class Car extends Bicycle {
   int wheelCount = 4;
}
```

Example 1.9. Constructor Chain

```
public class A {...}

public class B extends A {
   B(){ super(); }
}

public class D extends B {
   D(int n){ super(); }
   D(){ this(3); }
}

public class E extends D {
   E(){ //...
   }
}
```

3.3 Inheritance Violations

This category deals with wrong definitions or uses of inheritance. We considered two well known semantic free misuses of inheritance:

– **Inheritance to Extend Values (iev)** Liberman et al. in [18] report that inheritance can be mistakenly used to "extend values". For instance, some students think that inheritance can be used to change the values of fields, rather than for adding attributes or operations. Students with this kind of misconception can write code that resembles the excerpt shown in Example 1.8. A variable, `wheelCount`, with the same name of the superclass (`Bicycle`) is added into the subclass (`Car`). In fact, there is no overriding but the classes have two different variables named `wheelCount`, one in `Bicycle` and one in `Car`. Moreover, class `Car` has two variables named `wheelCount`—its own and the one inherited from `Bicycle`.
– **Constructor Chaining (cc)** The study in [18] also reports "that many students fail to understand the chain of constructor calls in object creation". A consequence of this failure could be producing the code in Example 1.9. Class `E` does not select which constructor of the superclass to use, and thus the default constructor is chosen—with probably unintended outcomes. 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 [1].

3.4 Interaction Violations

For what concerns interaction violations, our model considers the following two cases:

- **Unused Private Method (upm)** Method `doC()` in Example 1.10 is a private method which is never called. This is linked with "difficulties with scope", namely with the "private" keyword.
- **Static Invocation Through Instance (siti)** In Example 1.11, method `doSmthStat()` is static and so should be accessed in a static way, but is accessed by means of `this.doSmthStat()`. This is connected with what reported in [24] regarding "understanding `this` as the current object and its usage".

3.5 Polymorphism Violations

Polymorphism is managed, at source code level, with explicit casting. Our model cover the two major mistakes made by novices:

- **Wrong Explicit downCast** Students may think that "all down-casts in an inheritance hierarchy are legal" [18] and this lead to downcasting `R` to `S`—which is wrong, as shown in Example 1.12.
- **Unneeded Explicit Cast** Another issue with polymorphism is using type-casting explicitly especially when is not needed at all [16], like the statement regarding `a` in Example 1.13. This is a bad practice and usually typical of "programmers who have not fully understood the object-oriented paradigm use conditional statements to simulate dynamic dispatch and late binding" [31].

Example 1.10. Unused Private Method

```
public class SomeClass{
   public String doA(){
         System.out.println("do A");
   }
   public int doB(int a){
         System.out.print("do B");
   }
   private void doC(){ //...
   }
}
```

Example 1.11. Static Invocation Through Instance

```
public class G {
   private static void doSmthStat(){
         //...
   }
}
public class ClassWithSITI {
   public wrongCall(String s){
         this.doSmthStat();
   }
}
```

Example 1.12. Wrong Explicit downCast

```
public class P extends R {
         //...
}
public class R extends S {
         //...
}
public class S extends Z {
         //...
}
public class Z{
         //...
}

public class M {
   public void doWEC(){
         S s = (S) new R();
   }
}
```

Example 1.13. Unneeded Explicit Cast

```
public class M {

  public void doSmth(){
   S s = (S) new R();
   float f = 2.0;
   int a = (int)f/1;
  }
}
```

Example 1.14. Unused Association

```
public class E { //...
}
public class D {
  private E ua = new E();
  public void uaMethod(){
    System.out.print("2019");
  }
}
```

3.6 Relationship Violations

It is reported that "an extension of the difficulty with writing a program that includes multiple classes is writing a program that includes linked cooperating classes" [33]. Such difficulty could have led to including associations which are never used like association to class `E` in Example 1.14. We call this violation "unused association (ua)".

3.7 Quality Metrics Violations

For what concerns quality of produced source code, SPT evaluates product metrics. The product metrics summarize intrinsic properties of software components (such as the internal complexity or the external coupling). We selected the following metrics from the Chidamber-Kemerer (CK) Object-Oriented (OO) metric suite [8]: WMC (Weighted Method per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), RFC (Response for a Class), LCOM (Lack of Cohesion in Methods), CE (Efferent Couplings), NPM (Number of Public Methods), LCOM3 (Lack of Cohesion in Methods). In order to compute these metrics, the CKJM tool developed by Spinellis [28] was integrated into SPT as a recipe module.

3.8 Student Profiling Tool

The purpose of the Student Profiling Tool (SPT), is two-fold: (*i*) it allows producing collective reports to be used as feedback to plan or tune new teaching strategies for the entire classroom; (*ii*) it gives information regarding the behaviour of a single student as a personal assessment tool: it provides timely and personalized feedback based on the source code produced and analyzed [32]. The underlying framework used to implement SPT, Student Profiling Framework (SPF), is shown in Fig. 1. SPF is language-agnostic and must be instantiated to be applied. We have chosen to focus on Java adopting Spoon to build a per-project Abstract Syntax Tree (AST) performing source code analysis [21]. Specifically, the Source code analysis repository contains the definitions of the violations (See Sect. 3) which are performed by the static source code analyzer as well as a set of clustering analyses (See Sect. 4.1). The instructor may filter the input code clustering the projects in categories and, thus, the analyses

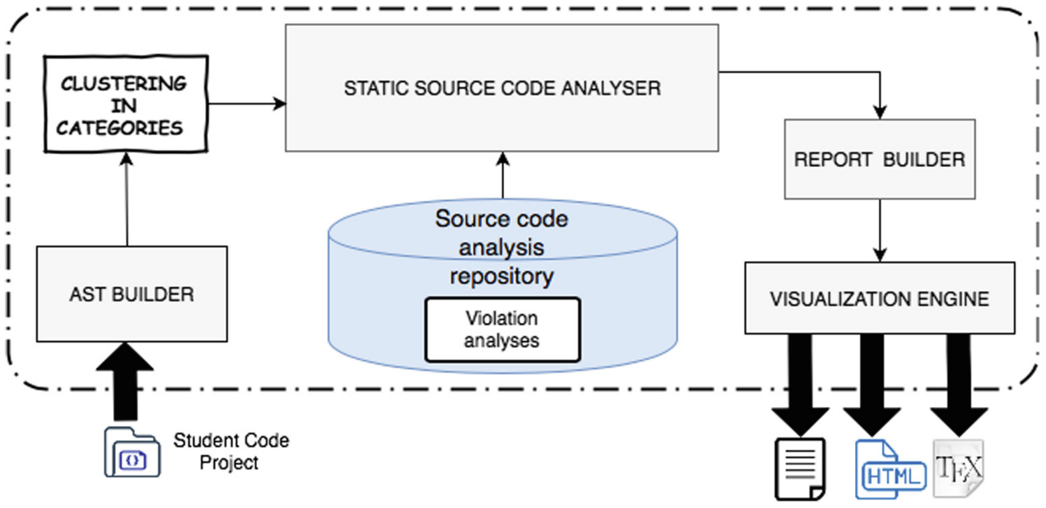
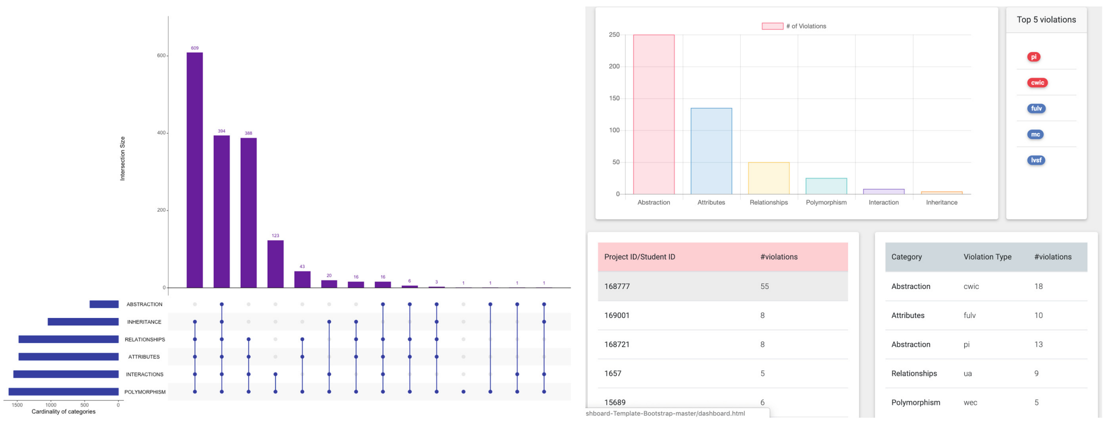


Fig. 1. Student profiling framework



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

shboard-Template-Bootstrap-master/dashboard.html

Category	Violation Type	#violations
Abstraction	cwic	18
Attributes	fulv	10
Abstraction	pi	13
Relationships	ua	9
Polymorphism	wec	5

Fig. 2. Distribution of the construct categories among the projects (left) and SPT dashboard (right).

of the language constructs which are not present in a given dataset will not be executed. Possible data-interchange and integration with existing software are available thanks to a custom visualisation engine which supports different types of output (raw text, HTML, LateX). Figure 2, on the right side, shows the results of a regular SPT session (some information are intentionally omitted and sample code is used). The instructor's dashboard gives an instant picture of the class, reporting statistics by category (top part) in a simple bar chart, top five violations (on the right) with matching color to understand the category, and two interactive tables showing the number of violations per student (bottom left) and their breakdown details (bottom right, i.e. "Category|Violation Type|#violations"). Therefore, the table on the left shows the particular situation for that student or student project ("168777" in the figure). By clicking on a particular violation type in the right table, `pi` for example, the instructor can visualize the description along with specific information for each violation ("Class|Related interface|Line" in case of `pi`). Instead, by choosing any row, corresponding Java code (with line numbers, syntax highlighting and injected comment) is presented with a marker on the line when violations occur. SPT also allows to create personalized PDF reports. Students will then receive a report with all their violations explained and suggested further reading. The latter is chosen by the instructor by modifying a configuration file (it includes lecture notes, open source books, specific books with chapter, pages and samples).

4 Empirical Study

The goal of this study, conducted exploiting SPT, is to investigate the volume of violations committed by students with the purpose of understanding which categories of language constructs are directly related. The quality focus is the understanding of object-oriented concepts and its relation to the students' ability to apply them. The perspective is mainly of researchers interested to investigate how, in student projects, miscomprehension of object-oriented language constructs of each category could favor errors on applying the object-oriented constructs of other categories. The perspective is also of teachers interested in improving their CS1 object-first courses by monitoring student performances. From this point of view, SPT allows identifying which constructs of the language are the most problematic within a class at a given time during the course and to study the evolution of the comprehension and ability to apply language constructs over time. The context of the study covers the data extracted from Blackbox [5]. BlackBox collects data from users of the online educational software tool called BlueJ[5]. The data collected is for academic research and addresses the issues related to teaching object-oriented languages. BlackBox has been running for over five years and contains a set of structural information (compiler data, code revisions, error messages) on over 12 million projects, 1.7 billion source history entries, and more than 2 billion events happened within the BlueJ environment.

4.1 The Context: Selection and Clustering

Due to the generous dimension of the dataset, it was necessary to perform a clustering step to obtain a set of projects suitable for the study. A set of ad-hoc scripts allowed to extract such projects by executing the following activities:

- selection of the successful compile events from all the events available in the repository;
- selection of projects with a number of files greater than two—to be able to study relationships, and lower than 15—to filter bigger projects;
- for all the compile events associated with files belonging to selected projects, creation of a CSV file with the following data: (i) identifier of the source code of a JAVA file, (ii) identifier of an event, (iii) successful compilation timestamp, and (iv) identifier of the project to which source code belongs to;
- for each project, using the above information to access index-payload[6] in the repository to extract project source code files used for source static analysis.

[5] https://www.bluej.org/.

[6] "payload files", at time of writing, go from Jun 12th 2013 to Oct 16th 2017; it was not possible to get the completed source code before and after this time interval.

Table 1. Top constructs violations in an excerpt of the dataset with projects pertaining all the categories.

N.	CATEGORY	NAME	#VIOLATIONS
1	Attributes	fulv	3164
2	Attributes	mc	2665
3	Abstraction	pi	2358
4	Attributes	lvsf	504
5	Attributes	pfc	352
6	Inheritance	cc	286
7	Attributes	upf	169

These steps led to a dataset of 1627 projects [9]. The dataset contains source code concerning different language constructs. Thus, there is no assurance a given project can be analyzed for all the considered categories. For this reason, SPT performs project clustering. For each considered project, it executes a static source code analysis to detect the categories of language constructs that are included in that project. The subsequent project analysis finds, for each category identified by the clustering step for that project, the violations described in Sect. 3.

The left side of Fig. 2 shows the results of the clustering. It reports the number of projects (vertical columns) contained in each combination of intersections (pointed lines) of the set of projects associated to each category of language constructs (shown in the horizontal lines on the bottom left). A dark dot in a row means that the corresponding set participates to the intersection. Otherwise, it means that such a set is excluded. For conciseness, Fig. 2 omits empty intersections and shows only the relevant combinations of categories. Looking at the figure, each project can belong to multiple categories and so create an intersection. For each intersection, we can evaluate the number of associated projects, i.e. the ones containing language constructs belonging to that category.

For example, 422 projects contain the *Abstraction* category. An interesting set contains all the six categories of language constructs (Abstraction, Inheritance, Relationship, Attributes, Interactions, and Polymorphism) and is comprised of 394 projects out of the total (1627). This set is essential since it contains a well balanced set of projects (of different sizes) and, for this reason, it is the reference set used as context of the empirical study.

4.2 Top Categories and Violations

The most violated categories, across all the projects in the dataset (1627), are Abstraction (42,17%) and Attributes (38,4%), followed by Relationships. The other categories have a very low rate—less than 6%. The Inheritance is the less infringed construct. Moreover, the dominant ratio of inheritance mistakes has a semantic root whereas in this study we are detecting language construct violations. Our results show that a small fraction of students makes semantic-free errors when applying inheritance. Another interesting perspective is the analysis

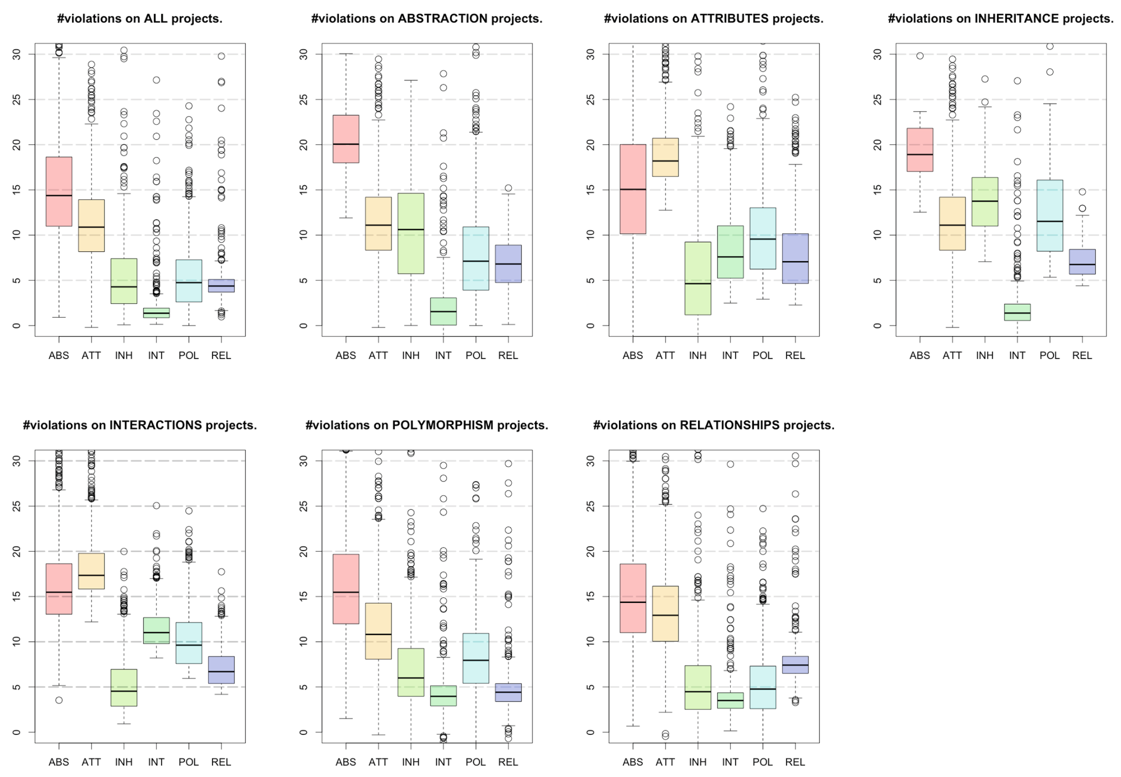


Fig. 3. Boxplots of the violations distributions for the considered categories of language constructs.

of top seven violations reported in Table 1, concerning projects belonging to all the categories.

The most present category is *Attributes*. Students seem to have problems with several aspects of handling attributes related to the kind of declared variables, i.e. most violations are related to instance fields that are defined in place of constants or used as a local variable, suggesting the lack of understanding of the meaning of instance variables. Also shadowing among variables is shown as problematical. Such a violation stands in the fourth place. Even this does not necessarily lead to a bug it suggests a lack in the comprehension of variable scoping rules. Violations of Abstraction are at third place in the top seven, for poor interfaces definitions. It seems students are not able to define interfaces accurately and, even when interfaces are well declared, they often miss the proper usage preferring the definition of variables using concrete types. This leads to worse modularization, increased coupling [30] and the usage of unneeded typecasting [16].

5 Results and Discussion

To address **RQ1**, we consider all the projects extracted from the blackbox dataset belonging to all the categories. We evaluate the distributions of number of violations on these categories using boxplots, as shown in Fig. 3—top-left. For each category, the figure shows the number of violations distribution, the inter-quartile range, and median value. The median number of violations of the projects follows a trend that is similar to the one we have already seen for

Table 2. Comparison of violation count distributions with Mann-Whitney test (✓ for $p-$value < 0.001 - ∅ otherwise) and cliff's delta d (medium and large values are highlighted in blue).

#violations distribution →	ABST	ATTR	INHE	INTE	POLY	RELA
ABST	✓ d=0.60	∅ d=0.03	✓ d=0.55	∅ d=-0.01	✓ d=0.29	✓ d=0.58
ATTR	∅ d=0.04	✓ d=0.80	∅ d=0.04	✓ d=0.95	✓ d=0.54	✓ d=0.58
INHE	✓ d=0.50	∅ d=0.03	✓ d=0.85	∅ d=0.01	✓ d=0.75	✓ d=0.79
INTE	✓ d=0.12	✓ d=0.77	∅ d=0.06	✓ d=0.97	✓ d=0.71	✓ d=0.77
POLY	✓ d=0.11	∅ d=-0.01	✓ d=0.22	✓ d=0.41	✓ d=0.76	∅ d=-0.01
RELA	∅ d=0.01	✓ d=0.25	∅ d=-0.01	✓ d=0.78	∅ d=0.01	✓ d=0.86
Projects of classes ↑						

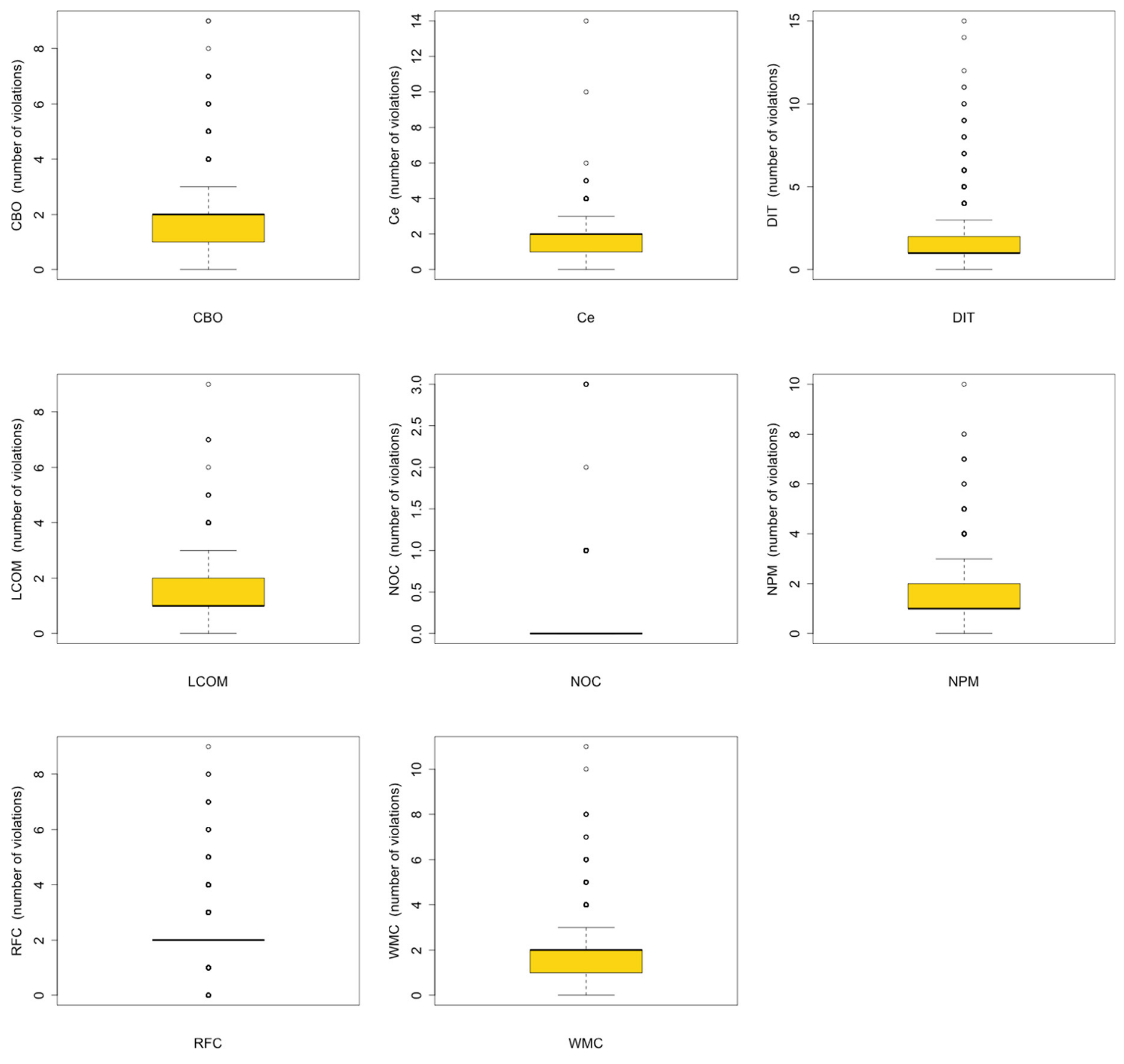


Fig. 4. Boxplots of the violations distributions for the considered CK metrics.

the total violations. In fact, Abstraction (∼15 violations), Attributes (∼13 violations) are the most infringed categories in descending order. The remaining categories (Relationships, Interactions, Inheritance and Polymorphism) have all a median number of violations that is less than five with inheritance being the lowest (with almost three violations per project). The second research question aims at understanding relationships among language constructs that are interrelated and, for this reason, cannot be easily taught or learned in isolation. To address **RQ2**, we consider the projects violating each category in turn and compute the number of violations on the remaining categories. We compare—using boxplots, Mann-Whitney test, and Cliff's delta—the above number of violations with the number of violations distribution evaluated on all the projects already calculated for **RQ1**. We then perform a pairwise comparison applying Mann-Whitney test and correcting p-values using the Holm's correction procedure [14]. This procedure sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by n − 1, and so on. Finally, in addition to the statistical comparison, we compute the effect size of the difference using Cliff's delta non-parametric effect size measure [13], defined as the probability that a randomly selected member of one sample has a higher response than a randomly selected member of the second sample, minus the reverse probability. Cliff's delta is considered negligible for $|d| < 0.147$, small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$. Figure 3 and Table 2 report the results to answer **RQ2**.

The boxplots show the comparison of the number of violation distributions evaluated for projects violating only one category with the one evaluated for all the projects. The table shows the pair-wise comparison using the Mann-Whitney test and Cliff's delta as a measure of the effect size. As Fig. 3 highlights, projects violating the Abstraction category have a significantly higher distribution (with respect to all the projects) of number of violations for *Inheritance*, *Polymorphism*, and *Relationships* language constructs. The effect sizes confirm all the relations (large for *Inheritance* and *Relationships*, and small for *Polymorphism*). The third plot in the first row of Fig. 3 shows that projects violating the *Attributes* category have a significantly higher number of violations, with respect to all the projects in the dataset, for *Interactions* (with a large effect size of d = 0.95) and, with a still large but lower effect size, for *Polymorphism* (d = 0.54) and *Relationships* (d = 0.58). This means that students having misconceptions about attributes and variable handling are more prone to commit errors that are related to object interactions, handling relationships among classes, and applying polymorphism correctly. With regards to projects violating *Inheritance*, results highlight a higher violation count for *Polymorphism* and *Relationships* with a large effect size in both cases.

For what concerning product metrics, boxplots of violations distributions are reported in Fig. 4. As figure shows, violations are mostly related to coupling problems (Ce, CBO and LCOM, NPM). Right behind coupling issues, we found complexity (WMC) and (DIT) with a comparable number of violations.

6 Conclusion and Future Work

We have presented an empirical study concerning 1627 projects of the Blackbox dataset [5]. The focus is on Java language constructs and their use.

We created a list of violations of language constructs, organized into seven categories and supported by existing literature in computer science/software engineering education. Next, we developed a tool, SPT, to do static analysis of Java student code and perform aforementioned empirical study.

The tool [9] can give an instant picture of the trend of a class and their learning process. Overall, students authored many violations with top three belonging to the categories of Abstraction and Attributes. To understand the relations between diverse categories we executed a clustering step (available through SPT). We believe our findings may be beneficial for harmonizing teaching strategies and designing new educational tools. Future work will focus on extending the list of violations, integrating STP into existing IDEs and also providing an interface for students to have new insights.

References

1. Ala-Mutka, K.M.: A survey of automated assessment approaches for programming assignments. Comput. Sci. Educ. **15**(2), 83–102 (2005)
2. Ardimento, P., Cimitile, M., Visaggio, G.: Distributed software development with knowledge experience packages. In: Demey, Y.T., Panetto, H. (eds.) OTM 2013. LNCS, vol. 8186, pp. 263–273. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41033-8_35
3. Biddle, R., Tempero, E.: Java pitfalls for beginners. ACM SIGCSE Bull. **30**(2), 48–52 (1998)
4. Brown, N.C.C., Altadmri, A.: Novice Java programming mistakes: large-scale data vs. educator beliefs. ACM Trans. Comput. Educ. (TOCE) **17**(2), 7 (2017)
5. Brown, N.C.C., AlTadmri, A., Sentance, S., Kölling, M.: Blackbox, five years on: an evaluation of a large-scale programming data collection project. In: ACM Conference on International Computing Education Research, ICER 2018, Espoo, Finland, 13–15 August 2018, pp. 196–204 (2018)
6. Buckers, T., et al.: UAV: warnings from multiple automated static analysis tools at a glance. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 472–476 (2017)
7. Chen, C., Cheng, S., Lin, J.M.: A study of misconceptions and missing conceptions of novice Java programmers. In: International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS), p. 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) (2012)
8. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
9. CSELAB. Student profiling tool (2018). https://gitlab.com/cselab/spt
10. Denny, P., Luxton-Reilly, A., Tempero, E., Ralph, P.: Objects count so count objects! In: Conference on International Computing Education Research, pp. 187–195. ACM (2018)
11. Edwards, S.H., Kandru, N., Rajagopa, M.B.M.: Investigating static analysis errors in student Java programs. In: ACM Conference on International Computing Education Research, ICER 2017, pp. 65–73. ACM, New York (2017)
12. Ettles, A., Luxton-Reilly, A., Denny, P.: Common logic errors made by novice programmers. In: Australasian Computing Education Conference, ACE 2018, pp. 83–89. ACM, New York (2018)
13. Grissom, R.J., Kim, J.J.: Effect sizes for research: a broad practical approach, 2nd edn. Lawrence Earlbaum Associates (2005)
14. Holm, S.: A simple sequentially rejective Bonferroni test procedure. Scand. J. Stat. **6**, 65–70 (1979)

15. Hristova, M., Misra, A., Rutter, M., Mercuri, R.: Identifying and correcting Java programming errors for introductory computer science students. SIGCSE Bull. **35**(1), 153–156 (2003)
16. Bergin, J., Agarwal, A., Agarwal, K.: Some deficiencies of C++ in teaching CS1 and CS2. ACM SIGPlan Not. **38**(6), 9–13 (2003)
17. Keuning, H., Heeren, B., Jeuring, J.: Code quality issues in student programs. In: ACM Conference on Innovation and Technology in Computer Science Education, pp. 110–115. ACM (2017)
18. Liberman, N., Beeri, C., Kolikant, Y.B.: Difficulties in learning inheritance and polymorphism. Trans. Comput. Educ. **11**(1), 4:1–4:23 (2011)
19. Luxton-Reilly, A., Denny, P., Kirk, D., Tempero, E., Yu, S.Y.: On the differences between correct student solutions. In: ACM Conference on Innovation and Technology in Computer Science Education, pp. 177–182. ACM (2013)
20. Madden, M., Chambers, D.: Evaluation of student attitudes to learning the Java language. In: Conference on the Principles and Practice of Programming, PPPJ 2002/IRE 2002, Maynooth, County Kildare, Ireland, Ireland, pp. 125–130. National University of Ireland (2002)
21. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: a library for implementing analyses and transformations of Java source code. Softw. Pract. Exp. **46**, 1155–1179 (2015)
22. Ragonis, N., Ben-Ari, M.: Teaching constructors: a difficult multiple choice. In: European Conference on Object-Oriented Programming, Workshop, vol. 3. Citeseer (2002)
23. Ragonis, N., Ben-Ari, M.: A long-term investigation of the comprehension of OOP concepts by novices. Comput. Sci. Educ. **15**(3), 203–221 (2005)
24. Ragonis, N., Shmallo, R.: On the (mis)understanding of the this reference. In: ACM SIGCSE Technical Symposium on Computer Science Education, pp. 489–494. ACM (2017)
25. Sanders, K., Thomas, L.: Checklists for grading object-oriented CS1 programs: concepts and misconceptions. In: Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2007, Dundee, Scotland, UK, 25–27 June 2007, pp. 166–170 (2007)
26. Schmolitzky, A.: Objects first, interfaces next or interfaces before inheritance. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 64–67. ACM (2004)
27. Sivilotti, P.A.G., Lang, M.: Interfaces first (and foremost) with Java. In: ACM Technical Symposium on Computer Science Education, pp. 515–519. ACM (2010)
28. Spinellis, D.: Tool writing: a forgotten art? (software tools). IEEE Softw. **22**(4), 9–11 (2005)
29. Tempero, E., Counsell, S., Noble, J.: An empirical study of overriding in open source Java. In: Australasian Conference on Computer Science, vol. 102, pp. 3–12. Australian Computer Society Inc (2010)
30. Tempero, E., Ralph, P.: A framework for defining coupling metrics. Sci. Comput. Program. **166**, 214–230 (2018)
31. Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: Jdeodorant: identification and removal of type-checking bad smells. In: 12th European Conference on Software Maintenance and Reengineering, CSMR 2008, pp. 329–331. IEEE (2008)
32. Wiggins, G.: Seven keys to effective feedback. Educ. Leadersh. **70**(1), 11–16 (2012)
33. Xinogalos, S.: Object-oriented design and programming: an investigation of novices' conceptions on objects and classes. ACM Trans. Comput. Educ. (TOCE) **15**(3), 13 (2015)