Most Common Fixes Students Use To Improve The Correctness Of Their Programs

Draylson Micael de Souza
University of São Paulo
São Carlos, São Paulo, Brazil
E-mail: draylson@icmc.usp.br

Michael Kölling
King's College London
London, England, UK
E-mail: michael.kolling@kcl.ac.uk

Ellen Francine Barbosa
University of São Paulo
São Carlos, São Paulo, Brazil
E-mail: francine@icmc.usp.br

***Abstract*—Teach students how to program is the main goal of most introductory CS courses. In fact, programming is one of the basic skills a professional in CS should have. However, there are many difficulties students face when they are learning how to program and, consequently, it is common introductory programming courses have high dropout rates. The purpose of this paper is to identify and discuss the most common fixes students use to improve the correctness of their programs. The findings can be useful to help students to produce more correct programs and highlight issues about possible difficulties they are having. To do so, we used the BLACKBOX data collection, which stores the actions of the BLUEJ programming environment users. The main idea was to observe the modifications students did in their source codes that made a failed JUNIT test case become succeeded. The results suggest the majority of fixes students use in their source codes are related either to the change of expressions or to the restructuring of code, reflecting difficulties in logic and problem solving among students.**

I. INTRODUCTION

Learning how to program has become increasingly important. Telling a machine what to do allows people to automate many repetitive tasks, making them more productive. By creating their own programs, people can receive more personalized results and obtain exactly what they need.

Despite the importance of programming, there are many difficulties students face when they have to learn how to program. In general, the problems range from a lack of understanding of basic programming concepts [1], [6], [2] to the lack of motivation to perform programming activities [3], [4].

Several studies have been conducted to understand the main difficulties and mistakes in programming. Some of them point out difficulties in learning to program based on students and instructors opinions [5], [6], [1]. Other studies discuss their findings through records of additional support sessions with TAs [7]. Research comparing the students' performance in programming and their knowledge in math and logic is also present in the literature [8]. The assessment of the students' mental model in contrast to their performance in programming is also being investigated [9], [10], [11]. In addition, some initiatives compare the quality of the students' programs and their actions when developing them [12]. Usually, correctness is one of the criteria considered for assessing the quality of the programs.

Thus, correctness of students' programs is an important subject to be studied. In the context of programming courses, correctness is usually assessed as the ability of the students' programs to produce the expected output to a given input data [13]. Know what increases the correctness of students' programs is useful to easily prevent or identify possible mistakes students made in their programs. For instance, if students always fix the same kind of bug to improve the correctness of their programs, activities which train them to easily prevent or identify those kind of bugs could help students to produce more correct programs.

This study aims to identify the most common fixes students use to increase the correctness of their programs. We considered the BLACKBOX data collection [14], which stores students' actions when they are programming on the BLUEJ development environment [15]. We observed projects containing JUNIT tests [16] and analyzed the fixes students used in their source codes that changed the status of a test from error/failed to succeeded. In general, the most common fixes identified are related either to the change of expressions or to the restructuring of code, suggesting difficulties in logic and problem solving among students.

The remainder of this paper is organized as following. Section II summarizes related work. Section III describes the method and procedures we considered to conduct our research. In Section IV we present the obtained results and in Section V we provide some discussions on them. Section VI summarizes our conclusions and future work.

II. RELATED WORK

In this section, we summarize the main related work to our study. Goals range from the identification of common bugs and fixes in programs to the broad understanding of the students' difficulties in programming.

Piteira and Costa [1] conducted a survey for students and teachers in programming. The survey included questions about the perception of the respondents about how problematic certain issues in programming and learning programming are, as well as about the difficulties related to each kind of programming concept and supporting materials.

Bryce et. al. [7] developed an online system to assist students with practical programming assignments in CS courses. Through the system, students can describe the bugs in their

978-1-5090-5920-1/17/$31.00 ©2017 IEEE

TABLE I: Comparison between our study and related works

Related Work	Both this study and the related work aim to identify	Related work obtain its results through	This study obtain its results through
Altadmiri and Brown [19]	Students' mistakes	Compilation errors	Wrong output errors
Edwards et. al. [12]	Relations between students' actions and the quality of their programs	Submitting events	Editing and testing events
Pan et. al. [18]	Common coding fixes	Open source projects	Novice programmers' projects
Vipindeep and Jalote [17]	Common coding bugs	Published articles and users' experience	Novice programmers' projects
Bryce at al. [7]	Difficulties in programming	Tutoring records	Editing and testing histories
Piteira and Costa [1]	Difficulties in programming	Surveys	Editing and testing histories

programs and tutors can provide support to them. Thus, the authors stored the students' bugs and the tutors' solutions. Then, they classified the bugs, according to the tutors solutions, in order to identify the main difficulties of the students.

Vipindeep and Jalote [17] created a list of common bugs found in programs based on published articles and users' experience in programming.

Pan et. al. [18] classified and analyzed the syntactic changes made in fixing faults of open source software projects implemented in Java.

Edwards et. al. [12] analyzed a set of data about students' programming assignments submissions. They collected the data through the use of Web-CAT system in CS programming courses. The authors compared the scores the students achieved and certain related data, such as number of submissions, time of first submission, time of last submission, total elapsed time and amount of code written.

Altadmiri and Brown [19] investigated the novice programming mistakes that causes compilation errors. They considered the BLACKBOX data collection which stores students action when they are programming on the BLUEJ development environment.

Table I summarizes the comparison between our study and related works. Our study follows a similar direction to that of Altadmiri and Brown's work [19], but focus on other aspects of the programming activity. While Altadmiri and Brown's work focuses on problems that cause compilation errors, we are focusing on programs which compile, but produce wrong outputs.

As in Edwards et. al.'s work [12], we are analyzing the quality of the students' programs, specially the correctness, but comparing it with the students' actions when they are editing and testing their programs instead of when they are submitting their programs to the assessment.

Similarly to Pan et. al.'s work [18], our idea is to classify and analyze the most common fixes performed to increase the correctness of the programs, but in an educational context instead of an open source development context.

Thus, we can obtain a list of common bugs, similar to Vipindeep and Jalote's work [17], but specific for novice programmers. Activities to train students to prevent or easier identify those kinds of bugs would help students to produce more correct programs.

Finally, the results can also give us more evidences about the students' difficulties in learning programming, complementing the results of Bryce at al. [7], Piteira and Costa [1], and others.

III. METHOD

Figure 1 illustrates the method we used to conduct our research. It is divided in three stages: (i) data collection, including procedures to collect the fixes the `BlueJ` users made to improve the correctness of their programs; (ii) classification, including the proposition of a classification schema to categorize the fixes; and (iii) data analyses, including procedures to take conclusions based on the data.

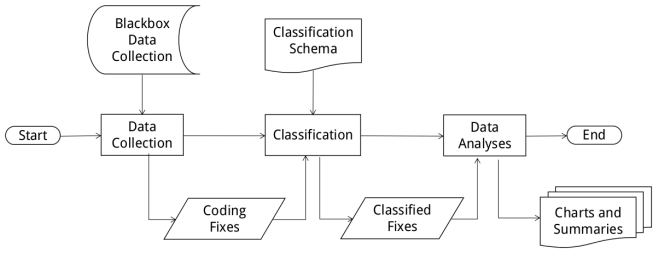


Fig. 1: Research Method

A. Data Collection

To address our goals, we have used BLACKBOX as our source of data. BLACKBOX [14] is a project associated with BLUEJ [15] – a programming development environment which focuses on the teaching and learning of object-oriented concepts through the Java programming language. In short, the BLUEJ main screen allows users to design the class structure of an application in an UML-like diagram. For each class, BLUEJ provides a code template which users can complete with their programming logic. BLUEJ allows students compile and execute their programs, instantiate arbitrary classes, invoke methods interactively, execute JUNIT tests, among other features.

BLACKBOX, in turn, is a data collection which stores the actions of the BLUEJ users. The data includes relevant details about the use of BLUEJ features, such as editing, compiling, execution, instantiation of objects, interactive method invocations, runs of unit tests, use of the debugger, use of source repositories, etc.

Two pieces of information should be collected from the BLACKBOX data in order to achieve our goals: (i) semantically incorrect program source codes; (ii) the fixes in the incorrect source codes which increased their correctness.

One of the issues we faced was the assessment of the source codes correctness. Since we did not have the specifications of the source codes, it was not possible to check whether their

executables' outputs were correct or not. To address this issue, we considered only the projects provided with JUNIT test cases. Thus, we considered the tests the students themselves used to assess the correctness of their programs. Although it is not possible state that a source code is correct when all tests are succeeded, it is possible to ensure that a test case failure indicates either the program source code is incorrect or the test case is incorrect, or both.

In this sense, we observed when a test changed its result from "error/failed" to "succeeded", considering these changes as our `samples`. Next, for each `sample`, we took the version of the source code which caused the test error/fail as the sample `incorrect source code` and the version of the source code which caused the test success as the sample `fixed source code`. Then, we took the differences between the `incorrect source code` and the `fixed source code`, considering them as the sample `fixes`. We discarded all samples containing modifications in a test in order to ensure we were considering only `samples` in which the program source code has incorrect.

Following this approach, we collected our `samples` considering all test runs performed in the year of 2015. Thus, we have observed 3,518,129 test runs across 18,810 projects. Then, we were able to identify 106,046 cases which a test changed its result from "error/failed" to "succeeded". Since many of them change their results due the same `fixes`, we got a total of 57,492 `samples` of `incorrect source codes` and their related `fixes`. All these samples came from students worldwide using BLUEJ and BLACKBOX.

We took the `fixes` by comparing the Abstract Syntax Trees (ASTs) of the source codes and we classified them according to a pre-defined classification schema which we detail in the following section.

B. Classification Schema

Aiming at addressing our goals, we proposed a classification schema which allows the accounting of the source code fixes in pre-defined categories. Thus, a single category contains two kinds of information: (i) the action performed to fix the source code; and (ii) the type of statement changed. Table II shows the four possible actions defined in our classification schema.

TABLE II: Classification Schema: Possible Actions

Actions	Description
Add (A)	The statement is present in the fixed source code, but not present in the incorrect source code.
Remove (R)	The statement is present in the incorrect source code, but not present in the fixed source code.
Move (M)	The statement is present in both source codes, but in different positions.
Change (C)	The statement was modified.

In order to define the types of statement in our classification schema we considered the Java Language Specification. Table III shows the 12 types of statements defined. The types were defined aiming at grouping the statements which are related to the same programming concept.

TABLE III: Classification Schema: Types of Statements

Types of statements	Description
Package Declaration (PD)	Statements related to the declaration of packages
Import Declaration (ID)	Statements related to the imports of external classes
Class Declaration (CD)	Statements related to the declaration of classes
Field Declaration (AD)	Statements related to the declaration of attributes
Method Declaration (MD)	Statements related to the declaration of methods
Variable Declaration Statement (VDS)	Statements related to the declaration of variables
Expression Statement (ES)	Expressions as statements, such as method calls and assignment statements
Selection Statement (SS)	`if-then`, `if-then-else` and `switch` statements
Loop Statement (LS)	`while`, `do` and `for` statements
Jump Statement (JS)	`break`, `continue` and `return` statements
Error Handling Statement (EHS)	`assert`, `try` and `throw` statements
Concurrency Control Statement (CCS)	`synchronized` statement

Considering the possible actions, for each type of statement we have four related categories. For instance, Table IV shows the categories related to the variable declaration statements.

TABLE IV: Classification Schema: Categories Related to the Variable Declaration Statements

Categories
Add Variable Declaration Statement (A-VDS)
Remove Variable Declaration Statement (R-VDS)
Move Variable Declaration Statement (M-VDS)
Change Variable Declaration Statement (C-VDS)

We also created subcategories to the categories related to the `Change (C)` action, classifying which changes were performed. Similarly to the top-level categories, each subcategory has two kinds of information: (i) the action performed; and (ii) the component of the statement changed. We considered the same four actions of the top-level categories and the possible components of the statements were also defined based on the Java Language Specification. According to each type of statement, the possible components are summarized in Table V. For instance, Table VI shows the subcategories of the `Change Variable Declaration Statement (C-VDS)`.

TABLE V: Classification Schema: Statements Components

Components	Description
Block (B)	Includes `else` block in a `if-then-else` statement, `finally` block in a `try` statement, etc.
Catch Clause (CC)	`catch` clause in a `try` statement
Dimension (D)	Array dimensions
Expression (E)	Used to affect the execution sequence in statements
Modifier (M)	Modifiers (such as `public`, `private`, `static`, `final`, etc.) in variables, fields, methods and class declarations
Switch Case (SC)	`case` clause in a `switch` statement
Type (T)	Variable types, method return value types, etc.
Type Parameter (TP)	Type parameters in generic classes
Variable Declaration (VD)	Declarations of variables in variable declaration statements, method declarations and `for` statements

TABLE VI: Classification Schema: Subcategories of the `Change Variable Declaration Statement (C-VDS)`

Subcategories
Add Modifier (C-VDS-A-M)
Remove Modifier (C-VDS-R-M)
Move Modifier (C-VDS-M-M)
Change Modifier (C-VDS-C-M)
Change Type (C-VDS-C-T)
Add Variable Declaration (C-VDS-A-VD)
Remove Variable Declaration (C-VDS-R-VD)
Move Variable Declaration (C-VDS-M-VD)
Change Variable Declaration (C-VDS-C-VD)

In addition to the above categories, we also observed the need to categorize the low-level changes in the expressions. Much of the work in a program is done by evaluating expressions. Their results affect the control flow and data flow of the program, being important in the determination the execution sequence in statements.

Thus, similarly to the statements, we defined types of expressions. Table VII shows the type of expressions defined in our classification schema. As an example, Table VIII shows the categories for the `Arithmetic Expressions`.

TABLE VII: Classification Schema: Types of Expressions

Type of expressions	Description
Arithmetic Expression (ARE)	Expressions related to arithmetic operators
Array Access (ARA)	Expressions related to the access to values in an array
Array Creation (ARC)	Expressions related to the instantiation of an array
Array Initializer (ARI)	Expressions related to the initialization of an array with a sequence of values
Assignment Expression (ASE)	Expressions related to assignment operators
Bitwise Expression (BTE)	Expressions related to bit wise operators
Cast Expression (CST)	Expressions related to cast operators
Class Instance Creation (CIC)	Expressions related to the instantiation of an class
Conditional Expression (CNE)	Expressions related to the Java conditional operator `(? :)`
Field Access (FLA)	Expressions related to the access to the value of a class field
Instanceof Expression (IOE)	Expressions related to the Java `instanceof` operator
Lambda Expression (LME)	Expressions related to the arrow token `->`
Literal (LTR)	Expressions related to literals such as `false`, `null`, `1`, etc.
Logical Expression (LGE)	Expressions related to logical operators
Method Invocation (MTI)	Expressions related to the invocation of methods
Method Reference (MER)	Expressions related to references to methods
Name (NAM)	Expressions related to variables, fields and objects
Parenthesized Expression (PRE)	Expressions related to parentheses to control the order of the expressions evaluation
Relational Expression (RLE)	Expressions related to relational operators
This Expression (THE)	Expressions related to the Java `this` keyword
Variable Declaration (VRD)	Declarations of variables in variable declaration statements, method declarations and `for` statements

Although variable declarations are not defined as expressions in the Java Language Specification, we considered them in our expression classification. Similarly to the `Assignment Expressions`, variable declarations play an important role in the program data flow, since they define and may assign values to local variables, fields and objects.

TABLE VIII: Classification Schema: Categories for the `Arithmetic Expressions`

Categories
Add Arithmetic Expression (A-ARE)
Remove Arithmetic Expression (R-ARE)
Move Arithmetic Expression (M-ARE)
Change Arithmetic Expression (C-ARE)

Finally, we defined subcategories for the expression categories related to the `Change (C)` action. The possible components of an expression can be: (i) an `Operator (O)`; (ii) statements components as `Dimension (D)`, `Modifier (M)` and `Type (T)`; and (iii) sub-expressions, including all types of expressions described above. Table IX shows examples of subcategories for the `Change Arithmetic Expression (C-ARE)`.

TABLE IX: Classification Schema: Subcategories for the Change Arithmetic Expression (C-ARE)

Subcategories
Change Operator (C-ARE-C-O)
Add Array Creation (C-ARE-A-ARC)
Remove Class Instance Creation (C-ARE-R-CIC)
Move Literal (C-ARE-M-LTR)
Change This Expression (C-ARE-C-THE)

C. Data Analyses

In this section, we describe the procedures we performed in order to analyze the data and take conclusions about them. Initially, we counted and analyzed the number of `diffs` between the `incorrect source codes` and the `fixed source codes` of each `sample`. The number of `diffs` can give us information about how much students modify their source codes in order to improve the correctness of their programs. To do so, we considered the textual differences of the source codes. Each line of source code added, removed or changed was counted as one `diff`.

After that, we conducted the analyses of the statements `fixes`. First, we counted and analyzed the number of fixes associated to each type of statement. The idea was find out what type of statement had more `fixes` associated with it.

Since the absolute frequency of `fixes` associated to a certain type of statement could be related to a high frequency of this type of statement in the source codes, we also analyzed the relative frequency of the statements `fixes`. In other words, we counted how many occurrences of each type of statement were in all `fixed source codes` of all `samples`. Then, we calculated the percentage of `fixes` associated to each type of statement in relation with the total of occurrences of the type of statement involved.

Then, we considered the categories proposed in our classification schema, analyzing the most common `fixes` associated to each type of statement and the most common statements `fixes` as a whole.

Finally, we conducted the analyses of the expressions `fixes`. We considered the same steps of the analyses of the statements `fixes`.

IV. Results

In this section, we discuss the obtained results from the study, according to the procedures previously described.

Table X summarizes the number of `samples` by the number of `diffs`. For instance, 2716 `samples` (7%) had 3 `diffs`, i.e. 3 lines were added, changed or removed from the `incorrect source code` to the `fixed source code`. Thus, 20213 `samples` (54%) had up to 3 `diffs`. The highest number of `diffs` was 1292, but 30087 `samples` (81%) had up to 13 `diffs`. Furthermore, 12754 `samples` (34%) had only one `diff`.

TABLE X: Number of `samples` by number of `diffs`

diffs	samples	samples (%)	samples (Cumulative)	samples (Cumulative %)
1	12754	34%	12754	34%
2	4743	12%	17497	47%
3	2716	7%	20213	54%
4	2263	6%	22476	60%
5	1615	4%	24091	64%
6	1119	3%	25210	67%
7	1044	2%	26254	70%
8	890	2%	27144	73%
9	822	2%	27966	75%
10	622	1%	28588	77%
11	548	1%	29136	78%
12	458	1%	29594	79%
13	493	1%	30087	81%
18	283	0,7%	31776	85%
27	123	0,3%	33427	90%
51	40	0,1%	35264	95%
1292	1	0,002%	37108	100%

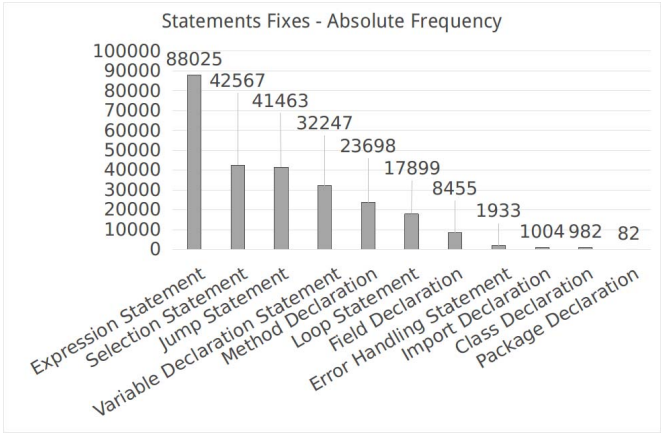
The results suggest the lower the number of `diffs`, the greater the number of `samples`. In other words, the results suggest that minor modifications are the majority of modifications students do in their source codes in order to increase the correctness of their programs.

A. Statements Analyses

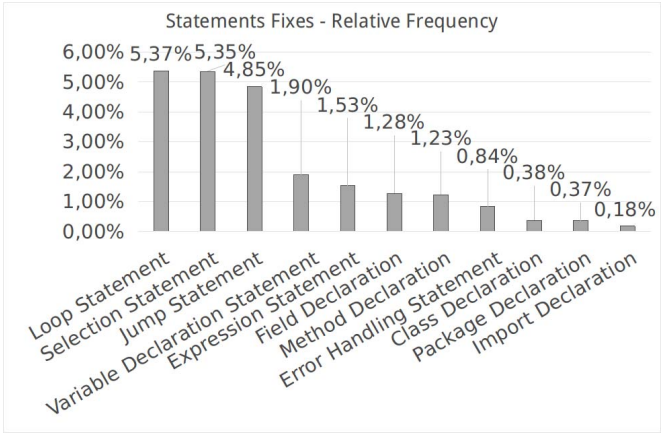
Figure 4a shows the absolute frequency of the statements fixes, `Expressions Statements` fixes are the most common, totaling 88025 fixes. Other common fixes were related to `Selection Statements` (42567 fixes) and `Jump Statements` (41463 fixes).

On the other hand, Figure 4b shows the relative frequency of the statements fixes, `Selection Statements` fixes are the most common, since the number of fixes in this type of statement in relation to the total of `Loop Statements` are 5.37%. Other common fixes are related to `Selection Statements` (5.35%) and `Jump Statements` (4.85%).

Comparing both absolute and relative frequencies, we can highlight the differences concerning the `Expression Statements` and `Loop Statements` fixes. The `Expression Statements` fixes has the highest number of occurrences, but it is less remarkable when considering the total of `Expression Statements`. That is, the `Loop Statements` fixes has a less remarkable number of occurrences, but this number is high when considering the total of `Loop Statements`.



(a) Absolute Frequency



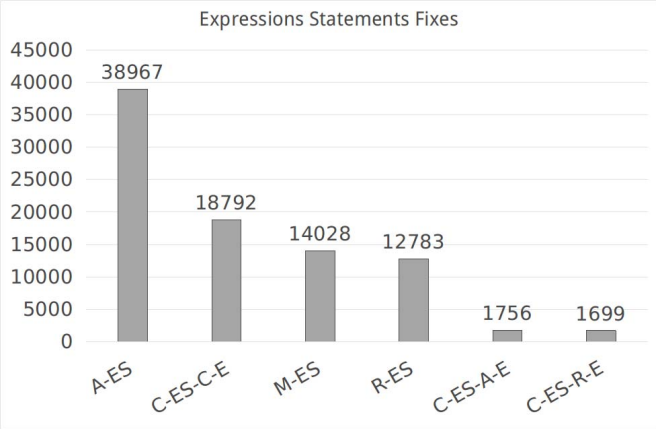
(b) Relative Frequency

Fig. 2: Statements Fixes

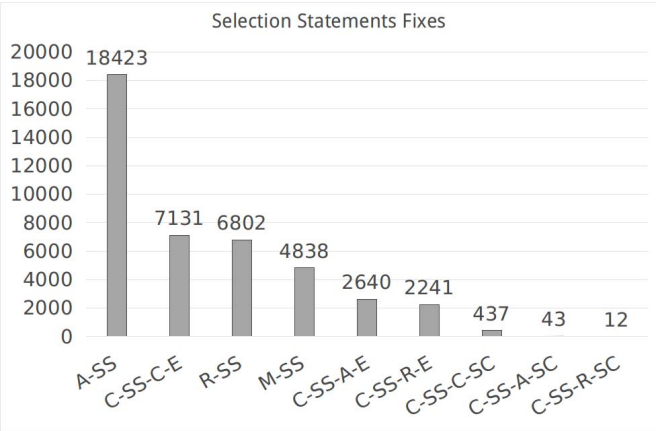
Therefore, the results suggest: (i) `Expression Statements` fixes are the most common, but they are also the most frequent type of statement in the programs; (ii) `Selection Statements` and `Jump Statements` fixes are among the most common, regardless the totals of these types of statements; and (iii) `Loop Statements` fixes are common if we take into account the total of `Loop Statements`.

Figure 3 details the results according to the categories and subcategories defined in our classification schema. For all types of statements, addition of new statements are the most common fixes (`A-ES`, `A-SS`, `A-JS` and `A-LS`). Then, change of expressions (`A-ES-C-E`, `A-SS-C-E`, `A-JS-C-E` and `A-LS-C-E`), movement of statements (`M-ES`, `M-SS`, `M-JS` and `M-LS`) and removal of statements (`R-ES`, `R-SS`, `R-JS` and `R-LS`) are the next three most common fixes, changing their positions according to the type of statement.

Therefore, the results suggest: (i) the most common fixes are related to the addition of new statements; (ii) fixes related to change of expressions are among the most common ones;

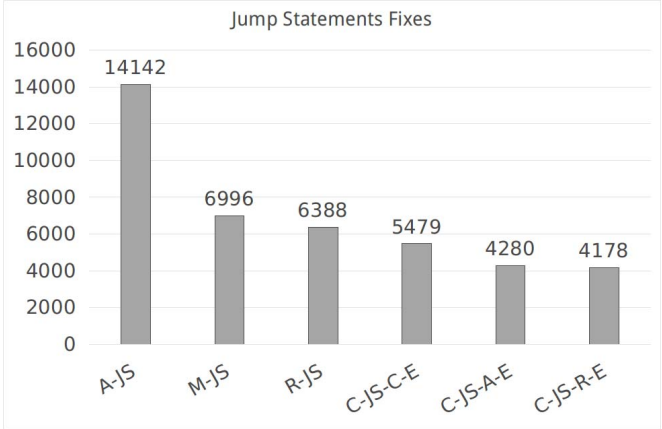


(a) Expression Statements Fixes

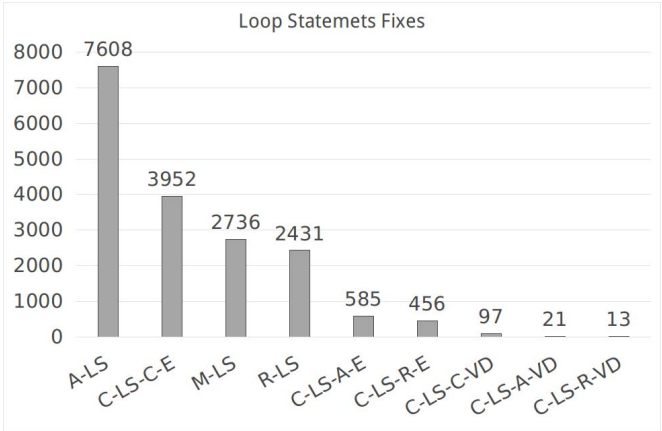


(b) Selection Statements Fixes

Fig. 3: Statements Fixes Categories and Subcategories



(c) Jump Statements Fixes



(d) Loop Statements Fixes

Fig. 3: Statements Fixes Categories and Subcategories (Cont.)

and (iii) fixes related to movement and removal of statements are also significant.

B. Expressions Analyses

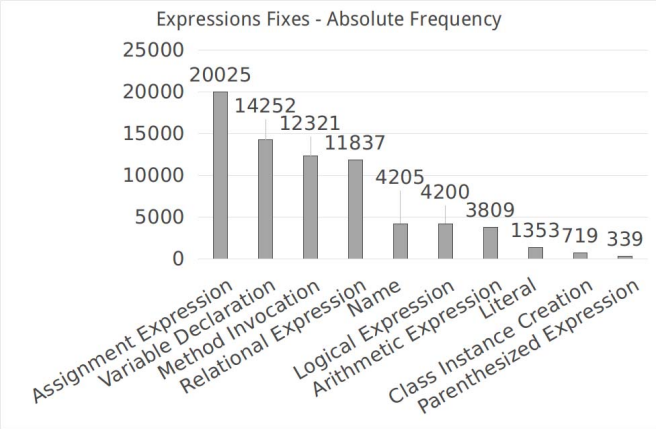
Figure 4a shows the absolute frequency of the expressions fixes. `Assignment Expressions` fixes are the most common, totaling 20025 fixes. Other common fixes were related to `Variable Declarations` (14252 fixes), `Method Invocations` (12321 fixes) and `Relational Expressions` (11837 fixes).

On the other hand, Figure 4b shows the relative frequency of the expressions fixes. `Logic Expressions` fixes are the most common, since the number of fixes in this type of expression in relation to the total of `Logic Expressions` are 0.77%. Other common fixes are related to `Relational Expressions` (0.71%) and `Assignment Expressions` (0.63%).

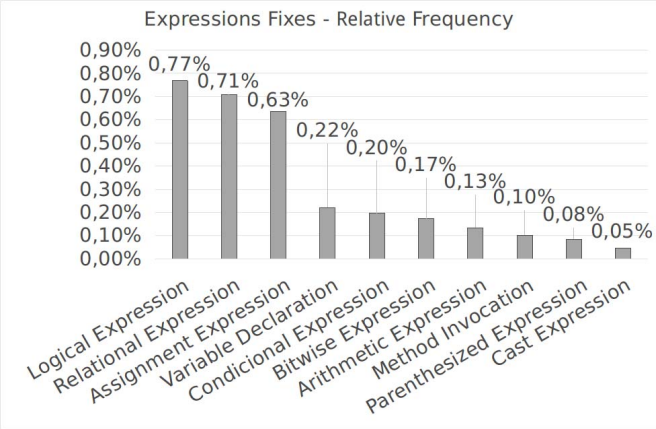
Comparing both absolute and relative frequencies, we can highlight the differences concerning expressions mostly related to data flow (such as `Assignment Expressions`, `Variable Declarations` and `Method Invocations`) and expressions mostly related to control flow (such as `Relational Expressions` and `Logic Expressions`). Fixes related to data flow expressions have higher number of occurrences, but they are less remarkable when considering their totals. On the other hand, fixes related to control flow expressions have lower number of occurrences, but they are more remarkable when considering their totals.

Therefore, the results suggest: (i) fixes in data flow expressions are common, but they are also frequent types of expressions; (ii) control flow expressions are common if we take into account their totals; (iii) `Assignment Expressions` and `Relational Expressions` fixes are among the most common, regardless the totals of these types of expressions.

Figure 5 and shows the results according to the categories and subcategories defined in our classification schema. In summary, `Change Name` (`C-ASE-C-NAM`, `C-VRD-C-NAM` and `C-MTI-C-NAM`) and `Change Arithmetic Expressions` (`C-ASE-C-ARE`, `C-VRD-C-ARE` and `C-MTI-C-ARE`) are the first and second most common fixes in data flow expressions, respectively. `Change Operator` fixes are the most common related to `Relational Expressions` (`C-RLE-C-O`) and `Change Relational Expression` fixes are the most common related to `Logical Expressions` (`C-RLG-C-RLE`), which



(a) Absolute Frequency



(b) Relative Frequency

Fig. 4: Expressions Fixes

reinforces the high number of fixes related to `Relational Expressions`.

Therefore, the results suggests: (i) the most common fixes related to data flow expressions are change of names and change of arithmetic expressions; (ii) the most common fixes related to control flow expressions are change of relational expressions, specially change of relational operators.

V. Discussion

Figure 6 summarizes the results and conclusions obtained through this study. Based on the results obtained and the observation of the incorrect source codes, we observed the bugs and their fixes are in three directions. The first is related to the high number of bugs whose fixes are related to the addition, movement and removal of statements. Looking at some examples, we verified, in several cases, students practically restructured their codes.

The second direction is related to a high number of bugs associated to the change of expressions in loop statements and selection statements. In fact, when performing the expression analysis, we identified a high number of bugs associated to relational expressions, especially, involving relational operators. Further, the most applied fixes in logical expressions were associated to the change of relational sub-expressions.

The third direction is related to bugs in expression statements. Through the expressions analysis, we observed a high number of fixes associated the change of arithmetic expressions in assignment expressions.

Comparing the results of this study with some difficulties in programming discussed in the literature, we could suggest the fixes involving a code restructuring are related to the difficulties of the students in the design of programs. In other words, students have difficulties in applying the several programming concepts in the construction of valid programs. These difficulties could be related to the lack of problem solving skills among students.

On the other hand, both bugs in relational expressions and bugs in arithmetic expressions could reflect the lack of knowledge in logic and mathematics among students, which leads to the difficulties in teaching and learning programming concepts.

VI. Conclusions

This study aimed to identify the most common fixes students use to increase the correctness of their programs. We analyzed the fixes students used in their source codes to change the status of their tests from error/failed to succeeded.

Based on the obtained results we can conclude the most common fixes are: (a) addition, movement and removal of statements in major source code modifications; and (b) change of expressions in minor source code modifications. Concerning the change of expressions, the most common fixes are: (i) change of fields, variables and objects names; (ii) change of relational expressions, specially change of relational operators; and (iii) change of arithmetic expressions in assignment expressions.

The fixes in major source code modifications can be related to difficulties among students in combine different programming statements to build a program that executes a certain task. On the other hand, the fixes in minor source code modifications can be related to difficulties among students in logic and

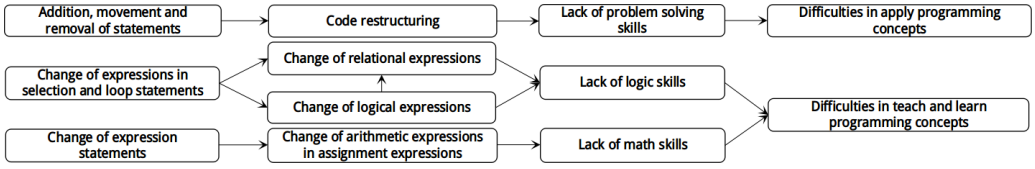
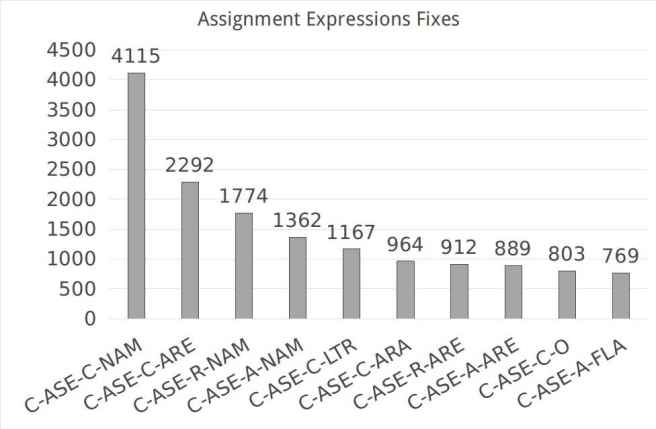
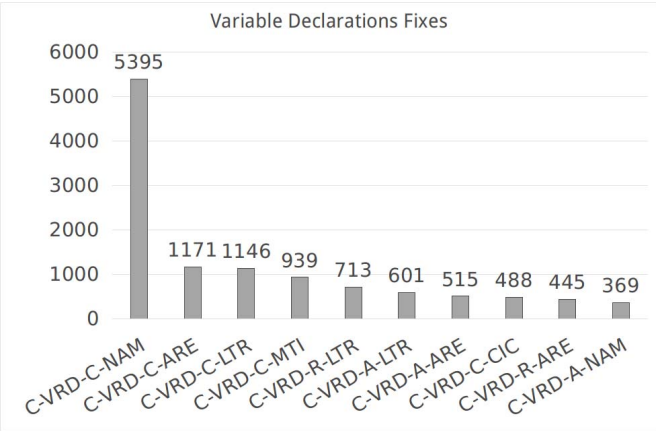


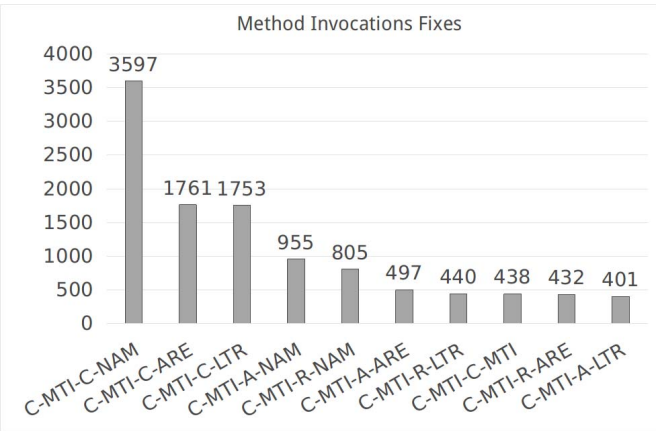
Fig. 6: Comparison between the bugs and difficulties in programming



(a) Assignment Expressions Fixes

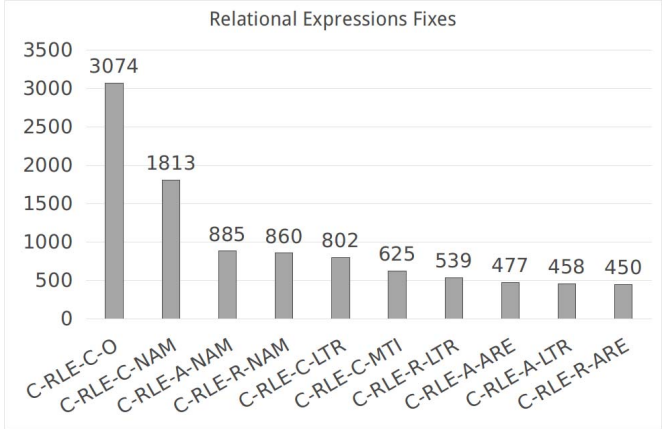


(b) Variable Declarations Fixes



(c) Method Invocations Fixes

Fig. 5: Expressions Fixes Categories and Subcategories



(d) Relational Expressions Fixes

(e) Logical Expressions Fixes

Fig. 5: Expressions Fixes Categories and Subcategories (Cont.)

math, especially involving concepts such as boundaries and conditions.

The results can be useful to easily prevent or identify possible mistakes students made in their programs. For instance, knowing that wrong arithmetic expressions in assignment statements are one of the common problems, we can create pedagogical strategies to better train students in arithmetic expressions and variable assignment. Another use would be the improvement of automated assessment tools in order to identify and provide a better feedback to students and instructors about the presence of those types of problems in students' programs.

As future work, we point out: (i) the proposition, execution and validation of pedagogical strategies and activities to train students in avoiding bugs associated to the identified fixes; and (ii) a more detailed study on the causes and difficulties in programming that can be leading to the high frequency of certain bugs and fixes. This study is already being designed and should be conducted in a short time.

ACKNOWLEDGMENT

The authors would like to thank the financial support of the Brazilian funding agencies: FAPESP (Proc. 2015/10670-9 and 2012/04352-6), CAPES and CNPq.

REFERENCES

[1] M. Piteira and C. Costa, "Learning computer programming: Study of difficulties in learning programming," in *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*, ser. ISDOC '13, 2013, pp. 75–80.

[2] C. Schulte and J. Bennedsen, "What do teachers teach in introductory programming?" in *Proceedings of the Second International Workshop on Computing Education Research*, ser. ICER '06, 2006, pp. 17–28.

[3] R. E. Anderson, M. D. Ernst, R. Ordóñez, P. Pham, and S. A. Wolfman, "Introductory programming meets the real world: Using real problems and data in cs1," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14, 2014, pp. 465–466.

[4] A. Settle, A. Vihavainen, and J. Sorva, "Three views on motivation and programming," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ser. ITiCSE '14, 2014, pp. 321–322.

[5] A. Alvarez and T. A. Scott, "Using student surveys in determining the difficulty of programming assignments," *J. Comput. Sci. Coll.*, vol. 26, no. 2, pp. 157–163, Dec. 2010.

[6] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," *SIGCSE Bull.*, vol. 37, no. 3, pp. 14–18, Jun. 2005.

[7] R. C. Bryce, A. Cooley, A. Hansen, and N. Hayrapetyan, "A one year empirical study of student programming bugs," in *2010 IEEE Frontiers in Education Conference (FIE)*, ser. FIE '10, 2010, pp. F1G–1–F1G–7.

[8] K.-Y. Lin, J. M.-C. Lin, and H.-C. Kao, "An analysis of difficulties encountered by novice alice programmers," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14, 2014, pp. 707–707.

[9] L. Ma, J. Ferguson, M. Roper, and M. Wood, "Investigating the viability of mental models held by novice programmers," *SIGCSE Bull.*, vol. 39, no. 1, pp. 499–503, Mar. 2007.

[10] C. Mirolo, "Mental models of recursive computations vs. recursive analysis in the problem domain," *SIGCSE Bull.*, vol. 41, no. 3, pp. 397–397, Jul. 2009.

[11] V. Ramalingam, D. LaBelle, and S. Wiedenbeck, "Self-efficacy and mental models in learning to program," *SIGCSE Bull.*, vol. 36, no. 3, pp. 171–175, Jun. 2004.

[12] S. H. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola, "Comparing effective and ineffective behaviors of student programmers," in *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ser. ICER '09, 2009, pp. 3–14.

[13] D. M. Souza, K. R. Felizardo, and E. F. Barbosa, "A systematic literature review of assessment tools for programming assignments," in *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, ser. CSEE&T '16, 2016, pp. 147–156.

[14] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14, 2014, pp. 223–228.

[15] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The bluej system and its pedagogy," *Computer Science Education*, vol. 13, no. 4, pp. 249–268, 2003.

[16] JUnit, "Junit cookbook," Available on: http://junit.org/junit4/cookbook.html, 2016, last access: July 7, 2017.

[17] V. V and P. Jalote, "List of common bugs and programming practices to avoid them," Available on: https://www.iiitd.edu.in/ jalote/papers/CommonBugs.pdf, 2005, last access: July 7, 2017.

[18] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, jun 2009.

[19] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15, 2015, pp. 522–527.