Understanding the relationships between self-regulated learning and students source code in a computer programming course

Hugo Castellanos, Felipe Restrepo-Calle, Fabio A. González
Department of Systems and Industrial Engineering
Universidad Nacional de Colombia
Bogotá, Colombia
Email: {hacastellanosm, ferestrepoca, fagonzalezo}@unal.edu.co

Jhon Jairo Ramírez Echeverry
Department of Electrical and Electronic Engineering
Universidad Nacional de Colombia
Bogotá, Colombia
Email: jjramireze@unal.edu.co

***Abstract*—To increase the success in computer programming courses, it is important to understand the learning process and common difficulties faced by students. Although several studies have investigated possible relationships between students performance and self-regulated learning characteristics in computer programming courses, little attention has been given to the source code produced by students in this regard. Such source code might contain valuable information about their learning process, specially in a context where practical programming assignments are frequent and students should write source code constantly during the course. This paper presents a strategy to support the correlation analysis among students performance, motivation, use of learning strategies, and source code metrics in computer programming courses. A comprehensive case study is presented to evaluate the proposed strategy through collected data (self-regulated learning characteristics and source code) from 205 undergrad students that accepted to participate voluntarily in the study during three semesters. Results show that the main features from source code which are significantly related to students performance and self-regulated learning features are: length-related metrics, with mainly positive correlations; and Halstead complexity measures, correlated negatively.**

I. Introduction

The rise in the use of computer systems in several fields has made computer programming an important subject in many engineering areas. Such importance has grown together with the improvement of computer systems and its use, and the interest of the governments to attract students to Science Technology Engineering and Mathematics (STEM) careers [1]. Computer programming requires some particular skills like: analyze and understand how a computer program works, understand at least one programming language, be able to design, implement, test and debug a computer program, among others [2].

Several authors have been focused on the better understanding of the learning and teaching processes in computer programming [3], [4]. Many approaches that consider students' source code to this matter are based on source code analysis tools. These are mainly aimed at generating automatic feedback [5], producing automatic assessments [6], and detecting plagiarism [7], [8] in computer programming assignments. However, information provided by this kind of tools is often related to the quality of the submitted assignment (i.e., source code of a computer program), without indicating students how to improve.

Regarding self-regulated learning studies in computer programming courses, several works have investigated possible relationships between students performance and self-regulated learning characteristics (i.e., motivation and learning strategies used by students) [9], [10], [11]. However, little attention has been given to another exclusive aspect from computer programming courses, i.e., students' source code (the students' work/output), with respect to its relationship with self-regulated learning. Current tools do not provide any information about these aspects.

Source code written by students might contain valuable information about their self-regulated learning process. This is particularly true in contexts where practical programming assignments are frequent and students write source code constantly during the course. Under these scenarios, instead of understand students' source code as a unique final result, it can be seen as another mean to monitor the learning process.

This poses the following research questions:

1) What is the relationship between the characteristics of students' source code and their performance in a computer programming course?
2) What is the relationship between source code features and self-regulated learning characteristics (i.e., motivation and learning strategies) in a computer programming course?

In order to answer these questions, this work presents a strategy to support the correlation analysis among students performance, motivation, use of learning strategies, and source code metrics in computer programming courses. A comprehensive case study is presented to evaluate the proposed strategy using data collected (self-regulated learning characteristics and source code) from 205 undergrad students that accepted to participate voluntarily in this study. They were enrolled in a Data Structures course (2nd-year course in Computer Science) at the *Universidad Nacional de Colombia*. Partici-

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

pants were distributed in three different semesters during 2015 and 2016. The course was supported by an automatic grading tool for programming assignments, namely *DomJudge*[1], which facilitated to obtain the source code of the participants for further automatic source code analysis. Moreover, self-regulated learning characteristics were collected using a Motivated Strategies for Learning Questionnaire [12], particularly the MSLQ-Colombia [13].

Results show that the main features from source code which are related to students performance in the course are: length metrics like lines of code, which is correlated positively; and Halstead complexity measures [14], which are correlated negatively. Regarding the relationship between source code features and self-regulated learning characteristics, results make evident significant correlations of Halstead complexity measures with the use of learning strategies such as: organization of ideas, and peer learning. In addition, there are interesting results showing correlations among learning strategies like: effort regulation and critical thinking, and source code metrics such as: length and complexity.

The paper is organized as follows: Section II presents the background with emphasis on self-regulated learning and source code metrics; in Section III, the proposed strategy is explained; in section IV the results are presented and discussed; finally, Section V presents the conclusions of this work.

II. Background and Related Works

A. Self-Regulated Learning

An important part of a student learning process is the self-regulation, which is defined as the process done to control cognition, behavior and motivation with the purpose of reaching a goal [15]. Among several models, the general structure of self-regulated learning from Pintrich [16] defines the possibility to self-regulate the following aspects:

- Cognition: to apply strategies to learn about a subject, including the construction of new concepts from previous knowledge.
- Motivation: including self-reward, and auto-persuasion with the purpose of improving interest in the subject.
- Behavior: to manage the learning resources, especially time, and in general, the actions which could help to improve the learning process, like study with peers.
- Environment: to generate strategies to adapt or control the environment, understanding this as the conditions in class, teacher behavior, etc.

Strategies must be created by the students in order to self-regulate which facilitates their learning process [17] to help to understand the reasons of failure or success. To be able to assess the self-regulation, the Motivated Strategies for Learning Questionnaire (MSLQ) is commonly used [12]. For the purposes of this work, the MSLQ-Colombia [13] adaptation was used. It is divided into two sections: motivation, and learning strategies. Measured by means of a set of features. These are described in Table I (motivation) and Table II (learning strategies). Each item in these tables has a number which is used to further identify the feature during the rest of this paper.

[1]http://www.domjudge.org

TABLE I: MSLQ Motivational features

#	**Motivational Features**
0	*Task value:* it is the value a student gives to a certain task, in terms of how interesting or how useful is the task.
1	*Anxiety:* includes students negative thoughts that affect performance.
2	*Extrinsic goals:* the perception to participate in a subject due to some kind of reward (grade competition).
3	*Control of learning beliefs:* the belief that the positive outcomes are the result of their own effort.
4	*Intrinsic goals:* the perception of the reasons why he/she likes the subject.
5	*Self-efficacy learning:* a self-evaluation of the ability to master a task.
6	*Self-efficacy performance:* it relates to the expectations about the performance in the course.

TABLE II: MSLQ Learning strategies features

#	**Learning Strategies Features**
7	*Time to study:* it is related to management, planning and effective use of the study time.
8	*Peer learning:* it refers to students learning with and from each other as fellow learners without any implied authority to any individual.
9	*Meta-cognition method:* continuous adjustment of the learning activities.
10	*Elaboration of ideas:* remember information in long-term memory by building connections among the items to learn.
11	*Effort regulation:* ability to control the effort avoiding distractions.
12	*Meta-cognition monitoring:* includes self-testing to be able to better understand the subject.
13	*Rehearsal:* repeating items in a list in order to memorize it.
14	*Critical thinking:* how the student apply old knowledge to deal with new situations.
15	*Study environment:* management and organization of the place where the person studies.
16	*Meta-cognition planning:* plan activities to ease the learning process.
17	*Organization of ideas:* to select and organize the information properly.

Self-regulation in computer programming learning in the academic context has been identified as an important issue. Several studies have explored self-regulation learning on computer programming courses. For instance, [18] studied motivational and self-regulation profiles adopted by students in computer science basic levels oriented to engineering. Moreover, [19] presented a study of the profile of 190 students in introductory programming courses with respect to their attitudes and self-regulation behavior. Results suggest that these groups of students have a very similar profile regarding learning strategies, self-efficacy perception, and organization of study activities. Also, studies found that vocational orientation to computer science-related careers is determined in many cases by self-efficacy and social support (like peer learning) [20]. In general, although motivation and learning strategies have similar traits among engineering students, these features have a strong personal and environmental component, and therefore, it is necessary to carry on specific studies on students self-regulated learning characteristics (i.e., motivation and learning strategies) [21].

Moreover, relationships between academic performance and self-regulation learning have been reported in several studies, such as [22], [23], [9], [10], [11]. Overall, good performance of programming students is characteristic of students with better self-regulated learning skills [24].

Taking into account the importance of self-regulation learning abilities in the students' performance, some tools have been developed to ease the self-regulation. For example, [25], [26] propose to foment these abilities through monitoring the progress of students in computer programming learning. These tools are oriented to teachers and students, which can use them to ease the planning, monitoring, and assessment of the learning process from both perspectives.

B. Source code metrics

Metrics are used in software source code to assess the quality of the product or process used to build it [27]. Such metrics have the following characteristics: quantitative (have a value), understandable, validatable, economical (metric extraction is not expensive), repeatable, language independent, applicable at any phase of the software development, comparable with another metric which measures the same concept. Metrics must have a scale which can be: interval (a defined range of values), ratio (value which has an absolute minimum or zero point), absolute, nominal (discrete scale of values, like 1-present or 0-not present) and ordinal (categories, i.e., levels of severity: critical, high, medium).

Source code metrics can be classified according to the intended measure:

- Size: intended to estimate cost and effort. The most popular metric of this category is the count of the source lines of code (SLOC).
- Software quality: intended to measure the quality of the software. This metric can be divided into the following categories:
  - Based on defects: these metrics attempt to measure the level of defects in a given time.
  - Usability: intended to measure the user satisfaction using the software.
  - Complexity metrics: oriented to produce a measure on the difficulty to test or maintain a piece of source code, e.g., the well-known McCabe complexity metric [28].
  - Halstead [14]: this group of empirical metrics is based on the count of operands and operators within the source code. They include measures to indicate: effort, time to understand the implementation, bugs delivered, among others.
  - Testing: intended to measure the progress of testing over a software.
- Object-oriented metrics: intended to measure object-oriented paradigm features. Among others, these include:
  - Coupling: measures the level of interdependence between classes. It is calculated counting the number of classes called by another class.
  - Cohesion: measures how many elements of a class are functionally related to each other.
  - Inheritance: it measures the depth of the class hierarchy.
  - Reuse: measures the number of times that a class is reused.
  - Size: intended to measure the size but not only in lines of code but also in the particularities of object-oriented paradigm, like method count, attribute count, class count, etc.
  - Evolutionary metrics: attempt to measure the evolution of a software based on elements like refactorings, bug-fixes.

Related works on students' source code analysis in an academic context include approximations with different purposes, such as: feedback [5] and assessment [6] support, cheat detection [7], [8], among others.

Some approaches are focused on automatic feedback and assessment, such tools have a test framework like unit test [29], acceptance tests defined in natural language-like scripts [30], etc. In addition, [31] suggests the creation of a submission policy which includes limit the number of submissions, amount of feedback received by the student, etc.

Techniques used in plagiarism detection approaches are similar to those used in information retrieval. For instance, the works presented in [32], [33] use N-grams to describe an author profile. Other approximations are based on the source code style, which commonly uses machine learning techniques together with a representation form. In [8], an Abstract Syntax Tree (AST) is used to represent the source code, including the style features. To identify authorship, they apply a random forest classification. In [34], [35] style features are extracted from the source code, and then a C4.5 algorithm is used to classify the author (students). In other works [36], besides the style information, they perform static analysis over the source code to use its output as another feature to identify the authors. The main advantage of these methods consists on the ease to understand and identify the discriminant features.

III. Exploring relationships between students' source code and self-regulated learning

The success rate improvement of computer programming courses is a source of concern for teachers. Understanding how students' motivation and learning strategies are correlated with the information available in their academic works could give a valuable insight to both teachers and students.

The purpose of the proposed strategy is to understand the relationship between source code metrics and performance, and at the same time, provide a new understanding of the relationship between source code metrics and students' moti-

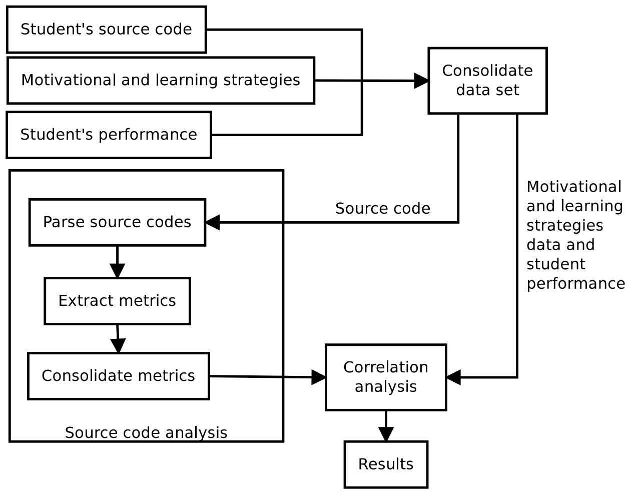

Fig. 1: General scheme of the proposed strategy

vation and learning strategies. Figure 1 presents the general scheme of the proposed strategy.

In the context of this work, there are three different data sources:

1) Students' source code: corresponds to each one of the attempts (correct or not) made by the students to solve an assignment in a computer programming course. Each attempt is processed no matter how much of the file is changed with respect to previous attempts, or if a previous attempt was already correct. Such assignments are worked and submitted throughout the course duration. The source code was collected from the *DomJudge* database, which was the automatic grading tool for programming assignments that supported the course.
2) Student performance: it is measured through the accumulative evaluation of the students, and it is represented using the final grade obtained by the student. This is the official grade provided by the teaching staff, which is based mainly on the results of the programming assignments. Grades are in the range 0.0 to 5.0, being 3.0 the minimum approval grade.
3) Motivation and learning strategies: corresponds to data which indicates the learning strategies used by the students and their motivation in a moment during the development of the course. This is obtained through the application of an MSLQ-Colombia test to the participant students [13]. This test contains 75 items, in which students specify their level of agreement or disagreement on a symmetric agree-disagree Likert scale from 1 (strongly disagree) to 7 (strongly agree).

Once the data sources are collected, they are consolidated in a single data set. This consolidation consists in the verification of the students who submitted programming assignments and answered the MSLQ test. This permits to label the students information as follows:

- Student name: to be sure the students are correctly identified as a unique person. This is specially important for those who in previous semesters dropped out or did not approve the course.
- Identify the real enrolled students: to be able to process only those students who were really enrolled to the course and discard those who were only attendants to the class, who could have submitted some of the assignments.
- Identify students who drop out the course: these students could have been enrolled in first place but later decided to cancel or leave the course. These students could have submitted some assignments and get some grades because of this, but they did not finish the course.
- Complete MSLQ survey responses: to be able to work with trusted data, it is verified that the students answered every single question from the MSLQ test. This is important because incomplete answers can generate unexpected results in further analysis stages.

Later, the source code analysis stage is performed. First, the consolidated data set of the students' source code is processed using a developed-from-scratch tool, designed to parse the source code files and to extract the source code metrics for each one of the files for every student. Table III presents the extracted source code metrics. These are classified into three main groups:

- *Length metrics*: contain the metrics related to some length/size measures. They are calculated as the average among: amount of files, average source lines of code, classes per file, source code lines per class, attributes per class, methods per class, class name length, and the average number of parameters.
- *Complexity metrics*: contain the metrics related to algorithm complexity. They are calculated as the average of: cyclomatic complexity, amount of for loops, amount of while loops, amount of if clauses, amount of if-else clauses, and the average identifier length.
- *Halstead*: contains all the Halstead metrics. They are calculated as the average of: Halstead bugs delivered, Halstead difficulty, Halstead effort, Halstead time to understand or implement, Halstead volume.

After the source code metrics have been extracted, the correlations analysis can be performed. Its purpose is to identify correlations between source code metrics, motivational features, use of learning strategies, and student performance. To properly calculate the correlation, a normal distribution test is done over the values. In case of normality, the Pearson product-moment correlation coefficient is calculated, otherwise, the Spearman correlation coefficient is used. This stage shows traces of possible relationships among the elements in the data sources.

The final step in the proposed strategy is the knowledge extraction by means of the use of machine learning techniques. A clustering process is done, and allows to confirm or discard the groups found in the correlations and to group the different features. Such groups also give information about the motivational features and the learning strategies used by the best-performing students.

TABLE III: Metrics calculated from the source code

#	**Metric**
0	Amount of files: total amount of files sent to *DomJudge*.
1	Average source lines of code: is the sum of all source code lines from all files divided by the number of files sent by a student.
2	Average class number by file: The total amount of classes in all source files divided by the number of files sent by a student.
3	Average source code lines by class: total amount of lines of all classes divided by the total class amount.
4	Average attributes by class: total number of attributes (static or not) divided by the total class amount.
5	Average methods by class: total number of methods (static or not) divided by the total class amount.
6	Average class name length: sum of the class name lengths divided by the number of classes.
7	Average amount of for loops: total amount of for loops divided by the number of methods.
8	Average amount of while loops: total amount of while loops divided by the number of methods.
9	Average amount of if clauses: total amount of if clauses divided by the number of methods.
10	Average amount of if-else clauses: total amount of if-else clauses divided by the number of methods.
11	Cyclomatic complexity: indicates the McCabe complexity metric
12	Average of static attributes: average of static attributes contained in a class
13	Average parameters: total number of parameters divided by the total number of methods in a class
14	Average of static methods: average of static methods per class
15	Average correct: average number of files which were correct according to the automatic grading tool
16	Average wrong: average number of files which were wrong according to the automatic grading tool
17	Average time limit: average number of files which hit time limit according to the automatic grading tool
18	Average compilation error: average number of files which had compilation error according to the automatic grading tool
19	Average execution error: average number of files which had execution errors according to the automatic grading tool
20	Average no output: average number of files which had no-output according to the automatic grading tool
21	Average identifier length: average identifier length by files, this includes names of variables, classes, parameters, etc.
22	Amount of correct files: total amount of correct files according to the automatic grading tool
23	Halstead - bugs delivered: Indicates the number of possible bugs generated
24	Halstead - Difficulty: an index that measures the difficulty
25	Halstead - Effort: an index that measures the necessary effort to write the source code
26	Halstead - Time to understand: an index which indicates the time taken to implement/understand the source code
27	Halstead - volume: indicates how much information the reader needs to get to understand the code

IV. Results and Discussion

For this experiment, a total of 205 undergrad students opted to participate voluntarily. They were enrolled in a Data Structures course (2nd-year course for Computer Science students) at the *Universidad Nacional de Colombia*. Students were distributed in three different semesters: 2015-I, 2015-II, and 2016-I. The duration of the course is 16 weeks.

A. Dataset descriptive statistics

Table IV summarizes the students' performance in the programming course. For the three semesters, this table shows the number of students who participated in the study (n), and some descriptive statistics for the final grade, i.e., mean ($\bar{x}$) and standard deviation (σ). Students are classified in three categories: approved, not approved, and dropout. Students whose final grade was ≥ 3.0 were classified as approved; those whose final grade was < 3.0 were classified as not approved; and students who did not have a final grade at the end of the course were categorized as drop out.

In average the approval rate of students among the three semesters was 59.8%. The failure rate (not approved) was 21.4%, and the drop out rate was 18.8%. As it can be seen, the drop out rate was higher in semester 2015-I than in the other two periods. In addition, the not approved rate was higher in semester 2016-I. Although this semester was the period with more students, the standard deviation in the final grade is lower than in the previous semesters for those who approved and slightly higher than 2015-II for those who did not.

Moreover, the motivational and learning strategies data was collected using MSLQ-Colombia [13], which is an adaptation of the original questionnaire to Colombia. Table V presents the descriptive statistics ($\bar{x}$ and σ) found for each one of the self-regulated learning features in the three academic periods. The questionnaire was applied in the fifth week of each one of the semesters.

In semesters 2015-I and 2015-II the standard deviation was less than 2.0 with small differences between semesters. However, in 2016-I it became higher, which can be explained by the number of students during this semester (101 students), almost twice the amount of students from other semesters. Furthermore, among the three semesters, some features have a significant difference between them. For instance, the average in critical thinking in 2015-I is 4.54, while in 2016-I it is only 3.71; organization of ideas and rehearsal, had their biggest averages (4.09 and 4.7 respectively) in 2015-II, and in 2016-I these values decreased to 2.41 and 2.92, respectively. Moreover, the semester 2015-II presents the lowest averages in motivational features. For example, the intrinsic goals is, in average, 3.06, which is low for a self-motivation feature. Nevertheless, this semester shows the best approval rate.

Finally, the dataset is completed with all the source codes submitted by each one of the students during the courses. Each row of the dataset will correspond to a single student. The dataset columns contain the MSLQ features, the source code metrics, and student performance. All data is normalized and consolidated in such a way that a single row corresponds to a single student.

TABLE IV: Students performance in the programming course

	2015-I				**2015-II**				**2016-I**			
	n	%	**Grade**		**n**	%	**Grade**		**n**	%	**Grade**	
			$\bar{x}$	σ			$\bar{x}$	σ			$\bar{x}$	σ
Approved	28	52.8	3.58	0.53	35	68.6	3.58	0.60	59	58.1	3.69	0.43
Not approved	9	16.9	1.13	0.93	7	13.7	2.00	0.74	34	33.6	1.97	0.8
Drop out	16	30.1			9	17.6			8	7.9		
Total	53				51				101			

TABLE V: MSLQ Data set description

	Feature	2015-I		2015-II		2016-I	
		$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
Motivation	Task value	5.78	1.39	4.39	1.82	4.76	2.65
	Anxiety	4.64	1.53	3.44	1.91	3.58	2.25
	Extrinsic goals	4.77	1.42	3.39	1.81	3.62	2.38
	Control of learning beliefs	5.79	1.25	4.19	1.79	4.83	2.69
	Intrinsic goals	4.50	1.55	3.06	1.61	3.89	2.38
	Self efficacy learning	5.44	1.31	4.09	1.80	4.40	2.57
	Self-efficacy performance	5.45	1.38	3.97	1.89	4.41	2.48
Learning strategies	Time to study	3.80	1.56	4.86	1.84	3.16	2.12
	Peer learning	3.93	1.86	4.21	1.68	3.45	2.23
	Meta-cognition method	4.15	1.65	4.18	1.72	3.33	2.10
	Elaboration of ideas	4.47	1.41	4.18	1.73	3.77	2.27
	Effort regulation	5.01	1.34	4.54	1.75	3.96	2.28
	Meta-cognition monitoring	5.25	1.26	4.58	1.80	4.26	2.40
	Rehearsal	3.86	1.54	4.70	1.80	2.92	1.95
	Critical thinking	4.54	1.32	4.27	1.67	3.71	2.18
	Study environment	5.31	1.58	4.81	2.00	4.12	2.53
	Meta-cognition planning	4.50	1.57	4.13	1.60	3.46	2.12
	Organization of ideas	3.12	1.67	4.09	1.68	2.41	1.73

B. Hierarchical bi-clustering over source code metrics and MSLQ features

To facilitate the further analysis of correlations between source code metrics and MSLQ features, a bi-clustering algorithm was applied to this data. Clustering results are shown in Figure 2. This figure presents a biclustering analysis of the correlations by means of a heatmap (due to space constraints it is shown only the heatmap for 2015-I). At the left side of the heatmap is represented the clustering of source code metrics and a color indicating the group of metrics each row belongs according to the metric clustering result. On top of the figure, the clusters corresponding to the MSLQ features are presented. In addition, the numbers on the vertical axis represent the source code metrics (right side of the figure), which correspond to the numbers of the metrics in Table III. The horizontal axis represents the MSLQ features (bottom of the figure), which are represented by the corresponding numbers in Tables I and II. In addition, the heatmap contains four highlighted regions of correlations which will be discussed in detail in next subsection.

Clustering results show that several source code metrics tend to group together. For instance, this is the case in four of the Halstead metrics. Metrics related to length (like the *average number of lines of code*, and *average methods by class*, among others) are close to each other as well (see the gray color at the left side of heatmap in Fig 2). Therefore, the source code metrics were grouped as follows:

- Length metrics (Gray): the amount of files, average source lines of code, the average amount of classes by file, average lines by class, averages attributes by class, average methods by class, average static attributes, and average static methods.
- Complexity metrics (Purple): average amount of *for* and *while* loops, average amount of *if* and *if-else* clauses, cyclomatic complexity, average identifier length, average class name length, and method parameter average.
- DomJudge result (Orange): average number of correct, wrong, time-limit, compiler error, execution error, and no-output solutions.
- Halstead (Green): bugs delivered, difficulty, effort, time to understand or implement, and volume.

C. Correlation analysis

The correlation study is done over the source code metrics (technical features) and the MSLQ features. Results are shown by means of the (previously mentioned) heatmap in Figure 2. Positive and negative correlations are represented in blue and red colors, respectively. The darker the color is, the correlation is closer to 1.0 (dark blue) or -1.0 (dark red). In addition, it is worth noting that regions of interest are highlighted using colors in the heatmap seen in Fig. 2, which corresponds to the main areas which will be discussed below (black at left, green on top, red in the middle, and orange the remaining rectangle).

The black rectangle covers an area which contains mainly Length metrics, which corresponds to gray color on the left side. It also shows a positive correlation between these metrics and three self-regulation items: Control of learning beliefs (3), Study environment (15), and Organization of ideas (17). This may suggest that a student with source code characterized by high values in length-related metrics may believe that a positive outcome in the course depends of his/her own effort. Also, it could suggest a good management in the use of the aforementioned learning strategies.

The green region of interest is mainly composed of negative correlations involving two groups of metrics (i.e., Complexity and DomJudge). Complexity has a strong negative correlation with self-efficacy learning (5). This fact may suggest little self-evaluation of the ability to master the course tasks in students whose code present high values in the metrics related to Complexity. Furthermore, there are two other motivational features involved, i.e., Task value (0) and Self-efficacy performance (6). As these features give an indication of the course importance to the students, this may suggest these students need more motivation during the semester.

The red highlighted region shows mainly Halstead metrics which are mostly correlated positively with learning strategies features. Time of study (7), elaboration of ideas (10), effort regulation (11) and meta-cognition planning (16) have a positive correlation with Halstead metrics. However, peer learning (8) have a negative correlation with this metric. This is particularly interesting because it may be indicating that learning with and from peers improve the value of these metrics, creating simpler code (in Halstead metrics the lower the better).

Finally, the orange rectangle covers again correlations between metrics in the Length group and the same MSLQ features than the green region of interest. Unlike the green rectangle, in this case the correlation are mostly positive,

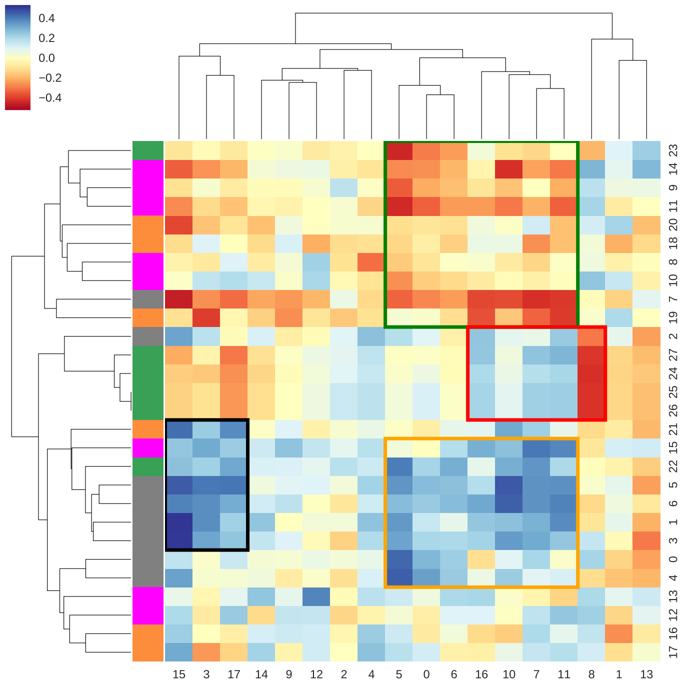

Fig. 2: Heatmap of correlations between source code metrics and self-regulated learning features in 2015-I. Groups of metrics are differenciated on the left: Length (Gray), Complexity (Purple), DomJudge (Orange), and Halstead (Green).

particularly in the case of self-efficacy learning (5). This may mean that length-related metrics indicate that students increase the value of this kind of metric due to practicing (sending more files/attempts) or being more explicit/verbose in code (e.g., long and explicative identifiers). This, in particular, should be further studied to confirm this relationship and also the importance of frequent programming practicing in this kind of courses. In addition, learning strategies features have mostly a positive correlation, which could indicate a dependency between the use of these learning methods and the length metrics in the source code.

In semester 2015-II, the correlations related to length technical features are not as strong as in semester 2015-I. Still, some interesting results can be mentioned. Length and Halstead metrics present significant (positive) correlations with several motivational features including extrinsic goals, control of learning beliefs, and self efficacy learning/performance. This is particularly interesting because it may be an indication about the improved quality of the source code according to the importance that the student gives to the class.

In semester 2016-I, the positive correlation in technical features 0 to 6 (length-related metrics) appear again but in this case correlated to the MSLQ features 9 to 14, which corresponds mainly to learning strategies. In the same semester, results show a negative correlation between several learning strategies from the MSLQ features against the source code metrics 7 to 15, which are mainly related with cyclomatic complexity. This indicates that cyclomatic complexity is negatively correlated to learning strategies like the effort regulation, elaboration of ideas and critical thinking. This might suggest that students exposing source codes with high cyclomatic complexity need to improve their learning strategies in order to improve their performance in relation to programming skills.

Next subsection discusses the correlations among source code metrics groups, MSLQ features, and students performance, including results for the three semesters.

D. Correlations of groups of source code metrics, motivational learning strategies, and students performance

Using the groups obtained for the source code metrics, the correlation coefficients were again calculated between these technical groups (source code metrics) and MSLQ features, including also students performance (i.e., final grade$-FG$). Results are presented in Table VI. The first column includes the number of the MSLQ feature (see Tables I and II), and also the final grade (FG). Due to space constraints, and to focus only on statistically significant results, presented correlations correspond only to those coefficients with a p-value ≤ 0.1. Correlations with an asterisk (*) in the Table corresponds more significant results where the p-value is ≤ 0.05. In this way, features and semesters without any significant correlation were removed from the Table.

As it can be seen in Table VI, source code metrics related to length are highly correlated with MSQL features and student performance in the course. In particular, effort regulation (11) and final grade (FG) show a positive correlation with the same tendency in two semesters (2015-I and 2016-I). These results suggest that length metrics have a correlation with the students performance in the class, which could be considered as a indicator to be monitored during the development of

TABLE VI: Correlation coefficients between MSLQ features (and students performance−*FG*) and groups of technical source code metrics

	Length			Compl.	DomJudge results			Halstead		
MSLQ	2015-I	2015-II	2016-I	2016-I	2015-I	2015-II	2016-I	2015-I	2015-II	2016-I
2			−0.22							
3		−0.27*			−0.33					
4							0.23*			−0.19
5	0.40*									
7	0.33							0.32		
8						0.30		−0.45*	0.27	
9				−0.21						
10	0.32									
11	0.30		0.17					0.33		
12		−0.28								
13			0.18							
14			0.23*	−0.22*						−0.19
15	0.53*									
17				−0.19				−0.32	0.38*	
FG	0.62*		0.44*							−0.30*

Correlations with an asterisk (*) means a p-value ≤ 0.05

programming courses. Moreover, this kind of metrics has the biggest number of significant correlations with motivational and learning strategies, many of them with p-value ≤ 0.05 (marked with *).

Moreover, Halstead metrics have during two semesters a significant correlation in peer learning (8) and organization of ideas (17). It can be seen that Halstead metrics have only one correlation with motivational features (4, intrinsic goals) and the rest are with learning strategies. This suggests that students exposing source code with high values in Halstead metrics (often associate to low quality code) may need to develop better learning strategies. This is consistent with the results observed in the Figure 2, which suggests that students whose source codes present high Halstead metrics need to improve their use of different learning strategies; for example, the organization of ideas.

The complexity metrics have a negative correlation with three learning strategies: meta-cognition method (9), critical thinking (14), and organization of ideas (17). As a high complexity metric is a bad indicator in software quality, this correlation may be understood as an indicator of which learning strategies a student needs to improve. Therefore, this could be a way to identify students which may need additional help. Furthermore, such improvements may lead to improving Halstead metrics which are also correlated with organization of ideas (17).

However, notice that the complexity metrics correlations appeared only in the semester 2016-I. As this was the semester with more students and higher percentage of not approved students (see Section IV-A), it is possible that bigger samples have influence in the final correlations. Additional studies are necessary to corroborate this and the relationships between MSLQ features and complexity metrics.

The DomJudge results present correlations with control of learning beliefs (3) in 2015-I, peer learning (8) in 2015-II, and intrinsic goals (4) in 2016-I, suggesting mainly a relationship with motivational features. It is consistent with the data shown in Table V, where control of learning beliefs obtained the biggest average in 2015-I, and the maximum peer learning result was obtained in 2015-II. Intrinsic goals, however, did not obtain the biggest average in 2016-I, but it might be caused due to the standard deviation, as in that semester it was higher than the others due to the large number of students in that semester.

Finally, students performance, i.e., final grade (*FG*) shows correlations with length and Halstead metrics. These correlations have a p-value ≤ 0.05, which is a strong indicator of the importance of those source code metrics in the students performance. As the correlation with Halstead metrics is negative, and the correlations with length metrics are positive, this could mean that good performing students write code with some of the following characteristics: easily readable, which causes a low value in Halstead metrics; with meaningful identifier names, which are more descriptive, causing a bigger value in length-related metrics. Therefore, measuring these metrics during the course could indicate which students may have problems to approve and may need additional assistance, which could be given based on other metrics. For instance, checking if the source codes have a high complexity, which suggests problems in some of the learning strategies.

V. Conclusions

Previous studies have focused in finding correlations between the self-regulated learning features and students performance in computer programming course. Although correlations were found in these studies, they provide few clues to give meaningful feedback to students on how to improve in their programming assignments in order to get better final grades. This paper provides an initial evidence that source code metrics in computer programming courses not only have correlations with the students performance but also with their self-regulated learning characteristics. This suggests that source code metrics could be a source of information about the students motivation and their adequate use of learning strategies. However, further investigations are necessary to explain the cause-effect relationships of these correlations.

In the light of the findings of this study, it is possible to understand better students' source code as an artifact that can be used to monitor several characteristics related to self-regulated learning, course performance, and in general, their learning process. In this way, more research in the area is required to verify if these relationships could give to computing educators new ways to identify and help those students with problems. In the future, the proposed strategy could be used as a base to build a tool which enables the teacher to give feedback to specific students early in the academic period.

As future work, the source code metrics considered in this work can be extended to other kinds. Due to the nature of the programming course studied in this paper, some metrics could not be applied. However, in more advanced courses, other metrics related to object-oriented programming could be considered, such as: coupling, cohesion, reuse, testing, etc.

References

[1] OCS, "Science, technology, engineering and mathematics in the national interest: A strategic approach," Canberra, 2013.

[2] M. Sahami and S. Roach, "Computer science curricula 2013 released," *Communications of the ACM*, vol. 57, no. 6, pp. 5–5, jun 2014.

[3] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, "A survey of literature on the teaching of introductory programming," *SIGCSE Bulletin*, vol. 39, no. 4, pp. 204–223, 2007.

[4] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[5] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 15–26, 2013.

[6] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, 2010, pp. 86–93.

[7] T. Bakker, "Plagiarism Detection in Source Code," Ph.D. dissertation, Universiteit Leiden, 2014.

[8] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing Programmers via Code Stylometry," 2014.

[9] S. Bergin, R. Reilly, and D. Traynor, "Examining the role of self-regulated learning on introductory programming performance," in *Proceedings of the first international workshop on Computing education research*. ACM, 2005, pp. 81–86.

[10] V. Kumar, P. Winne, A. Hadwin, J. Nesbit, D. Jamieson-Noel, T. Calvert, and B. Samin, "Effects of self-regulated learning in programming," in *Advanced Learning Technologies, 2005. ICALT 2005. Fifth IEEE International Conference on*. IEEE, 2005, pp. 383–387.

[11] C. S. Chen, "Self-regulated learning strategies and achievement in an introduction to information systems course," *Information technology, learning, and performance journal*, vol. 20, no. 1, p. 11, 2002.

[12] P. R. Pintrich, D. A. F. Smith, T. Garcia, and W. J. McKeachie, "A Manual for the Use of the Learning Questionnaire Motivated Strategies for (MSLQ)," *Mediterranean Journal of Social Sciences*, vol. 6, no. 1, pp. 156–164, 1991.

[13] J. J. Ramírez-Echeverry, A. García-Carrillo, and F. A. Olarte Dussán, "Adaptation and Validation of the Motivated Strategies for Learning Questionnaire -MSLQ- in Engineering Students in Colombia," *International Journal of Engineering Education*, vol. 32-4, 2016.

[14] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[15] M. Boekaerts, S. Maes, and P. Karoly, "Self-Regulation Across Domains of Applied Psychology: Is there an Emerging Consensus?" *Applied Psychology*, vol. 54, no. 2, pp. 149–154, 2005.

[16] P. R. Pintrich, "The role of motivation in promoting and sustaining self-regulated learning," *International Journal of Educational Research*, vol. 31, no. 6, pp. 459–470, 1999.

[17] C. E. Weinstein and R. E. Mayer, "The teaching of learning strategies." in *Innovation abstracts*, vol. 5, no. 32. ERIC, 1983, p. n32.

[18] K. G. Nelson, D. F. Shell, J. Husman, E. J. Fishman, and L.-K. Soh, "Motivational and self-regulated learning profiles of students taking a foundational engineering course," *Journal of Engineering Education*, vol. 104, no. 1, pp. 74–100, 2015.

[19] A. P. Ambrosio, L. Almeida, A. Franco, S. Martins, and F. Georges, "Assessment of self-regulated attitudes and behaviors of introductory programming students," in *Frontiers in Education Conference (FIE), 2012*. IEEE, 2012, pp. 1–6.

[20] M. B. Rosson, J. M. Carroll, and H. Sinha, "Orientation of undergraduates toward careers in the computer and information sciences: Gender, self-efficacy and social support," *ACM Transactions on Computing Education (TOCE)*, vol. 11, no. 3, p. 14, 2011.

[21] B. J. Zimmerman, "Developing self-fulfilling cycles of academic regulation: An analysis of exemplary instructional models." 1998.

[22] D. DiFrancesca, J. L. Nietfeld, and L. Cao, "A comparison of high and low achieving students on self-regulated learning variables," *Learning and Individual Differences*, vol. 45, pp. 228–236, 2016.

[23] C. Cheng and E. Keung, "The role of self-regulated learning in enhancing learning performance," 2011.

[24] S. Alhazbi, "Using e-journaling to improve self-regulated learning in introductory computer programming course," in *Global Engineering Education Conference (EDUCON), 2014 IEEE*. IEEE, 2014, pp. 352–356.

[25] M. Manso-Vázquez and M. Llamas-Nistal, "A monitoring system to ease self-regulated learning processes," *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, vol. 10, no. 2, pp. 52–59, 2015.

[26] O. Ortiz, P. M. Alcover, F. Sánchez, J. Á. Pastor, and R. Herrero, "M-learning tools: The development of programming skills in engineering degrees," *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, vol. 10, no. 3, pp. 86–91, 2015.

[27] R. Malhotra, *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*. CRC Press, 2015.

[28] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[29] M. Amelung, P. Forbrig, and D. Rösner, "Towards generic and flexible web services for e-assessment," in *ACM SIGCSE Bulletin*, vol. 40, no. 3. ACM, 2008, pp. 219–224.

[30] J. P. Sauvé and O. L. Abath Neto, "Teaching software development with atdd and easyaccept," *ACM SIGCSE Bulletin*, vol. 40, no. 1, pp. 542–546, 2008.

[31] L. Malmi, V. Karavirta, A. Korhonen, and J. Nikander, "Experiences on automatically assessed algorithm simulation exercises with different resubmission policies," *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, p. 7, 2005.

[32] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, "Source code author identification based on N-gram author profiles," *IFIP International Federation for Information Processing*, 2006.

[33] R. Layton, P. Watters, and R. Dazeley, "Local n-grams for author identification: Notebook for PAN at CLEF 2013," in *CEUR Workshop Proceedings*, vol. 1179, 2013.

[34] B. S. Elenbogen and N. Seliya, "Detecting Outsourced Student Programming Assignments," *Journal of Computing Sciences in Colleges*, vol. 23, no. 3, pp. 50–57, 2008.

[35] R. R. Joshi and R. V. Argiddi, "Author Identification : An Approach Based on Style Feature Metrics of Software Source Codes," vol. 4, no. 4, pp. 564–568, 2013.

[36] J. H. Hayes and J. Offutt, "Recognizing authors: An examination of the consistent programmer hypothesis," *Software Testing Verification and Reliability*, vol. 20, no. 4, pp. 329–356, 2010.