Enhancing Software Quality in Students' Programs

STELIOS XINOGALOS, University of Macedonia
MIRJANA IVANOVIĆ, University of Novi Sad

This paper focuses on enhancing software quality in students' programs. To this end, related work is reviewed and proposals for applying pedagogical software metrics in programming courses are presented. Specifically, we present the main advantages and disadvantages of using pedagogical software metrics, as well as some proposals for utilizing features already built in contemporary programming environments for familiarizing students with various software quality issues. Initial experiences on usage of software metrics in teaching programming courses and concluding remarks are also presented.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics – *Complexity measures*; K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*

General Terms: Education, Measurement

Additional Key Words and Phrases: Pedagogical Software Metrics, Quality of Students' Software Solutions, Assessments of Students' Programs

1. INTRODUCTION

Teaching and learning programming presents teachers and students respectively with several challenges. Students have to comprehend the basic algorithmic/programming constructs and concepts, acquire problem solving skills, learn the syntax of at least one programming language, and familiarize with the programming environment and the whole program development process. Moreover, students nowadays have to familiarize with the imperative and object-oriented programming techniques and utilize them appropriately. The numerous difficulties encountered by students regarding these issues have been recorded in the extended relevant literature. Considering time restrictions, large classes and increasing dropout rates the chances to add more important software development aspects in introductory programming courses, such as software quality aspects, seems to be a difficult mission.

On the other hand, several empirical studies regarding the development of real-world software systems have shown that 40% to 60% of the development resources are spent on testing, debugging and maintenance issues. It is clear both for the software industry and those teaching programming that the students should be educated to write code of better quality. Several efforts have been made from researchers and teachers towards achieving this goal. These efforts focus mainly on:

- adjusting widely accepted software quality metrics for use in a pedagogical context,
- devising special tools that carry out static code analysis of students' programs.

This paper focuses on studying the related work and making some proposals for dealing with software quality in students' programs. Specifically, we propose utilizing features already built in contemporary programming environments used in our courses, for presenting and familiarizing students with various software quality issues without extra cost. Of course, using pedagogical software metrics is not an issue that refers solely to pure programming courses. However, since students formulate their programming style in the context of introductory programming courses, it is important to introduce pedagogical software metrics in such courses and then extend on other software engineering, information systems and database courses, or generally in courses that require from students to develop software. The rest of the

This work was partially supported by the Serbian Ministry of Education, Science and Technological Development through project *Intelligent Techniques and Their Integration into Wide-Spectrum Decision Support*, no. OI174023 and by the Provincial Secretariat for Science and Technological Development through multilateral project "*Technology Enhanced Learning and Software Engineering*".
Authors' addresses: S. Xinogalos, Department of Applied Informatics, School of Information Sciences, 156 Egnatia str., 54006 Thessaloniki, Greece, email: stelios@uom.gr; M. Ivanović, Department of Mathematics and Informatics, Faculty of Sciences, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia, email: mira@dmi.uns.ac.rs

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

paper is organized as follows. In the 2nd section we refer to adequate related work. Section 3 considers usage of pedagogical software metrics. In section 4 we present some initial experiences of usage of software metrics in teaching programming courses. Last section brings concluding remarks.

2. RELATED WORK

When we refer to commercial software quality a long list of software metrics exists that includes basic metrics and more elaborated ones, as well as combinations and variations of them. Highly referenced basic metrics are: (i) the *Healstead metric* that is used mainly for estimating the programming effort of a software system in terms of the operators and operands used; and (ii) the *McCabe cyclomatic complexity measure* that analyzes the number of the different execution paths in the system in order to decide how complex, modular and maintainable it is.

The problem with such metrics is that they have not been developed for use in a pedagogical context. As [Patton and McGill 2006] state such metrics have the potential to be utilized for analysis of students' programs, but they have specific shortcomings: several metrics give emphasis on the length of the code irrespectively of its logic and do not differentiate between various uses of language features, such as a *for* versus a *while* loop, or a *switch-case* versus a sequence of *if* statements. When we talk about students' programs it is clear that as educators we consider the logic of a program more important than its length, while the appropriate utilization of language features is one of the main goals of introductory programming courses. In this sense, researchers have proposed software metrics specifically for analyzing student produced software.

One such framework has been proposed by [Patton and McGill, 2006] and includes the following elements: [1] *language vocabulary*: use of targeted language constructs and elements (e1); [2] *orthagonality/encapsulation*: of both tasks (e2) and data (e3); [3] *decomposition/modularization*: avoiding duplicates of code (e4) and overly nested constructs (e5); [4] *indirection and abstraction* (e6); [5] *polymorphism, inheritance and operator overloading* (e7).

Patton and McGill [2006] devised this framework in the context of a study regarding optimal use of students' software portfolios and propose attributing its elements to specific pedagogical objectives, and weighting them according to the desired outcomes of the institution and instructor.

Another recent study aimed at devising a list of metrics for measuring static quality of student code and at the same time utilizing it for measuring quality of code between first and second year students. In this study, seven code characteristics (in italics) that should be present in students' code are analyzed in 22 properties, as follows [Breuker et al. 2011]: [1] *size-balanced*: (p1) number of classes in a package; (p2) number of methods in a class; (p3) number of lines of code in a class; (p4) number of lines of code in a method, [2] *readable*: (p5) percentage of blank lines; (p6) percentage of (too) long lines, [3] *understandable*: (p7) percentage of comment lines; (p8) usage of multiple languages in identifier naming; (p9) percentage of short identifiers, [4] *structure*: (p10) maximum depth of inheritance; (p11) percentage of static variables; (p12) percentage of static methods; (p13) percentage of non-private attributes in a class, [5] *complexity*: (p14) maximum cyclomatic complexity at method level; (p15) maximum level of statement nesting at method level, [6] *code duplicates*: (p16) number of code duplicates; (p17) maximum size of code duplicates, [7] *ill-formed statements*: (p18) number of assignments in an 'if' or 'while' condition; (p19) number of 'switch' statements without 'default'; (p20) number of 'breaks' outside a 'switch' statement; (p21) number of methods with multiple 'returns'; (p22) number of hard-coded constants in expressions.

Some researchers have moved a step forward and have developed special tools that perform static analysis of students' code. Two characteristic examples are CAP [Schorsch 1995] and Expresso [Hristova et al. 2003]. *CAP* ("Code Analyzer for Pascal") analyzes programs that use a subset of Pascal and provides user-friendly and informative feedback for syntax, logic and style errors, while Expresso aims to assist novices writing Java programs in fixing syntax, semantic and logic errors, as well as contributing in acquiring better programming skills.

Several other tools have been developed with the aim of automatic grading of students' programs in order to provide them with immediate feedback, reducing the workload for instructors and also detecting

plagiarism [Pribela et al. 2008, Truong et al. 2004]. However, in most cases these environments are targeted to specific languages, such as CAP for Pascal and Expresso for Java. A platform and programming language independent approach is presented in [Pribela et al. 2012]. Specifically, the usage of software metrics in automated assessment is studied using two language independent tools: SMILE for calculating software metrics and Testovid for automated assessment.

However, none of these solutions has gained widespread acceptance. Our proposal is to utilize features of contemporary programming environments and tools in order to teach and familiarize students with important aspects of software quality, as well as help them acquire better programming habits and skills without extra cost. Usually features of this kind are not utilized appropriately, although they provide the chance to help students increase the quality of their programs easily.

3. USING PEDAGOGICAL SOFTWARE METRICS

3.1 Advantages and Disadvantages

Pedagogical software metrics can be applied with various ways in courses having a software aspect with the ultimate goal of developing better quality software. Specifically, they can be given to students just as guidelines to follow in order to develop quality code, or as factors that count towards grading their software products. In the latter case it is clear that a considerable amount of time should be devoted in training students in comprehending and applying the selected software metrics. On the one hand this is important to take place even from introductory programming courses, since this is the time when students formulate their "good" or "bad" programming style/habits that is not easy to change in the future. On the other hand, novices have several difficulties to deal with when introduced to programming and adding formal rules regarding software metrics might not be a good choice at least not for all students. Moreover, adding more material in introductory programming courses is not easy in terms of both time and volume of material.

Several researchers and instructors have integrated software metrics in systems used for automatic checking of software developed by students, either for grading their programs or/and for detecting plagiarism. The advantages are several. First of all, students can get immediate feedback about their achievements and be supported in overcoming their difficulties and misconceptions, while grading is fair. Secondly, instructors save a great deal of time from correcting programs, a process that in the case of large classes and many practical exercises is extremely time-consuming. Of course, developing such tools is also not easy and requires a great deal of time and effort.

3.2 The Educational IDE BlueJ

The educational programming environment BlueJ is a widely known environment used in introductory object oriented programming courses, since it offers several pedagogical features that assist novices. These features can be appropriately utilized for teaching and familiarizing students with software quality aspects described in the previous section and helping them acquire better programming habits.

Editor features. The editor of BlueJ provides some features that can help students firstly appreciate a good style of programming and secondly inspect their code for the existence of properties proposed in the framework by [Breuker et al., 2011] or the elements proposed by [Patton and McGill, 2006], or other similar frameworks. These features are:

- *line numbers* that can be used for a quick look on the lines of code in methods (p4) and classes (p3) if the instructor considers it important and provides students with relevant measures for a project
- *auto-indenting* and *bracket matching* help students write code that is better structured and more readable. However, several times students do not consider style so important and they write endless lines of code with no indentation and no distinction between blocks of code. In the case of errors that are so common in student's programs, this lack of structure makes the detection of errors difficult especially in the case of nested constructs (e5). The instructor can easily convey this concept to students by presenting students such a program (or using their own ones) and

using the *automatic-layout* ability provided for BlueJ for presenting them the corresponding program with proper indentation in order to help them realize the difference in practice.

- *syntax-highlighting* can help students easily inspect their code for ill-formed statements (p18-p21). However, the instructor has to make students comprehend that they have to inspect the code they write and not just compile and run it. Syntax-highlighting, for example, can help students easily detect a sequence of 'if' statements that should be replaced by an "if/else if.." or 'switch' construct.
- *scope highlighting* that is presented with the use of different background colors for blocks of code should be – in the same sense as above – utilized for a quick inspection of nested constructs (e5) and level of statement nesting at method level (p15) in order to avoid increased complexity. The instructor can give students some maximum values to have in mind and ring them a bell for reconsidering the decomposition/modularization of their solution.
- *method comments* can be easily added in students' code. When the cursor is in the context of a method and the student invokes the 'method comment' choice a template of a method comment is added in the source code containing the method's name, java doc tags and basic information regarding parameters, return types and so on. Students must understand that comments (p7) produce more readable and maintainable code and also can be used for producing a more comprehendible and valuable *documentation view* of class. This interface of a class is important in project teams and the development of real world software systems.

Moreover, if instructors think that a more formal approach should be adopted towards checking coding styles the BlueJ *CheckStyle* extension [Giles and Edwards] can be used. This extension is a wrapper for the CheckStyle (release v5.4) development tool and allows the specification of coding styles in an external file.

Incremental development and testing. Students tend to write large portions of code before they compile and test it, increasing this way the possibility for error-prone code of less quality. We consider that it is important to develop and test a program incrementally in order to achieve better quality code. BlueJ offers some unique possibilities for novices towards this direction. Specifically, the ability of *creating objects and calling methods with direct manipulation techniques* makes incremental development and testing an easy process. Students are encouraged to create instances of a class (by right-clicking on it from the simplified UML class diagram of a project presented in the main window of BlueJ) and call each method they implement for testing its correctness. Students can even call a method by passing it - with direct manipulation techniques - references to objects existing and depicted in the *object bench*. This makes incremental developing and testing of each method much easier and less time consuming. The invocation of methods should always be done with the *object inspector* of each object active, in order to check how the object's state is affected and also how it affects method execution. Students should be encouraged to use the *object inspector* to check: encapsulation of data (e3); static variables (p11); private and non-private attributes (p13). It is not unusual for students to write code mechanically and so it is important for them to learn to inspect afterwards what they have written. This also stands out for methods as well. The pop-up menu with the available methods for an object of a class, shows explicitly the public interface of a class and can help novices comprehend public and private access modifiers in practice and utilize them appropriately. Also, the dialog box that appears when a student creates an object or calls a method for an object, "asks" the student to enter a value of the appropriate type for each parameter and helps students realize whether their choices of parameters were correct (i.e. a parameter is missing or it is not needed). Students can experiment with all the aforementioned concepts by writing the corresponding statements in the *Code pad* incorporated in the main window of BlueJ.

Visualization of concepts. The main window of BlueJ presents students with a simplified *UML class diagram* giving an overview of a project's *structure*. Specifically, the following information is presented: name of each class; type of class (concrete, abstract, interface, applet, enum); 'uses' and 'inheritance' relation. This UML class diagram can be used for getting an overview of a project either it is given to students for studying it or it is developed by students themselves. Students can easily inspect the overall structure of a project, the number of classes (p1) and the depth of inheritance (p10). Students should also be encouraged to inspect the UML class diagram in order to: detect classes representing

related entities that have been defined independently and in this case refactor the project using inheritance; check for cohesion and coupling and validate the decisions made while coding. In this process, the *Class card extension* [Steinhuber, 2008] can support further students, since it can present (in a different card) for each class information regarding its attributes and methods.

Debugger. Finally, BlueJ offers a debugger that allows students to set a *breakpoint* in the desired source code line and execute the code in a *step by step* manner, watching the *call sequence stack*, inspecting the values of *static*, *instance* and *local variables*.

4. SOME EXPERIENCES ON USING PEDAGOGICAL SOFTWARE METRICS

At the Department of Technology Management at the University of Macedonia in Greece pedagogical software metrics were applied, until recently that the Department was merged with the Department of Applied Informatics, only in an informal way. The Department offered a 2nd semester "Introduction to Programming" course using C as the programming language and a 3rd semester "Object Oriented Design and Programming" course based on Java. At both courses students were presented with numerous examples of good style programs and were encouraged to use them as templates for developing their programs. In the introductory course emphasis was given on the common code style conventions, ill-formed statements, level of nesting and the merits of abstraction (mainly usage of functions). At the Object Oriented Programming course that students develop programs of considerable bigger size and complexity, emphasis was given – in addition – on good structure, encapsulation, code duplicates, coupling and cohesion. However, all these were part of presenting the various programming concepts to students during lectures and practicing with them in labs, and were not presented as software metrics per se. In labs the various features of BlueJ were utilized as described in the previous section in order to help students realize the importance and adopt a good style of programming, an incremental development and testing programming strategy, as well as the habit to always inspect the structure of their programs and test its correctness. Although there is no formal assessment, the author's experience on teaching the courses the last 8 years and grading students' programs developed as homework or during exams, has showed that the majority of students apply a good style of programming, fewer students adopt an incremental development and testing strategy and even fewer test them rigorously.

At the Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad students, within "Introduction to programming" course in 1st semester using Modula-2 as the programming language, were also presented with wide range of simple examples of good style programs. Students were, during theoretical and lab exercises, encouraged using them as templates for developing their good style programs. Additionally one topic within the course is devoted to essential elements of good-style programming and also emphasis was given on the common code style conventions, ill-formed statements, level of nesting and the merits of abstraction. Later in other programming courses, especially within 3rd semester "Object Oriented Programming" course based on Java, teachers insist that students use these good style programming elements intensively in their solutions.

For lab exercises within different programming courses at our Department teachers use specially developed framework Svetovid for submission and assessment of students' software solutions. Svetovid represents a good educational tool that can be further gradually developed and enhanced. Standard part of the framework is Testovid component devoted to automatic assessment of students programming solutions. Testovid allows students to test their assignments in a controlled manner and allows teachers to run the same tests on a set of students' assignments. The component accepts any type of files as assignment and the teacher has a great flexibility in specifying how and what aspects of students' solutions have to be tested (like coding style, successful compiling, and so on). Furthermore, Testovid incorporates hints and advices into the testing reports, enabling teachers to give students comprehendible feedback about their programming solutions [Pribela et al. 2012]. This feature of the system inspired us to try to incorporate also software metrics to provide students with rich information about fallacies of their solutions and additionally evaluate the quality of student programs. It represents a good starting point

[Pribela et al. 2012] in the direction of development of appropriate pedagogical software metrics and to utilize software metrics in the assessment process of the student solutions to programming assignments.

5. CONCLUSIONS

Nevertheless the fact that software metrics are a well known way to measure the quality of software, existing automated assessment systems that have adopted them are still rare. It is also questionable if widely speeded software metrics like: Halstead metrics, McCabe cyclomatic complexity and some other NDepend metrics are common and useful static metrics for computer science education purposes. Our initial attempts to apply software metrics in programming courses and gained experiences indicate that it is useful and even necessary to develop and apply in everyday teaching specific pedagogical software metrics. Usage of such metrics could help students to develop systematic and higher-quality programs starting from introductory programming courses and further through other programming courses during their faculty education. Such approach could prepare them for future jobs and real-life software development where application of software metrics is getting unavoidable. So, in our future work we are going to put significant effort on developing specific pedagogical software metrics.

REFERENCES

Dennis M. Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring static quality of student code. In *Proc. of the 16th annual joint conference on Innovation and technology in computer science education* (ITiCSE '11). ACM, New York, NY, USA, 13-17.

Rick Giles and Stephen H. Edwards. 2011. Checkstyle. Available online [Last access on 24 July 2013]: http://bluejcheckstyle.sourceforge.net/#overview

Arnold L. Patton and Monica McGill. 2006. Student portfolios and software quality metrics in computer science education. *J. Comput. Sci. Coll.* 21, 4 (April 2006), 42-48.

Ivan Pribela, Mirjana Ivanović, and Zoran Budimac. 2009. Svetovid – interactive development and submission system with prevention of academic collusion in computer programming, *British Journal of Educational Technology* 40, 6, 1076-1093.

Ivan Pribela, Gordana Rakić, and Zoran Budimac. 2012. First Experiences in Using Software Metrics in Automated Assesssment. In *Proc. of the 15th International Multiconference on Information Society (IS)*, 250-253.

Tom Schorsch. 1995. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. ACM *SIGSCE Bulletin* 27, 1, 168-172.

Michael Steinhuber. 2008. Class Card – A Better UML Extension. Available online [Last access on 24 July 2013]: http://klassenkarte.steinhuber.de/index_en.html

Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static analysis of students' Java programs. In *Proc. of the Sixth Australasian Conference on Computing Education - Volume 30* (ACE '04), Raymond Lister and Alison Young (Eds.), Vol. 30. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 317-325.