Annales Univ. Sci. Budapest., Sect. Comp. 31 (2009) 203-215

MEASURING THE COMPLEXITY OF STUDENTS' ASSIGNMENTS [1]

M. Juhás, Z. Juhász, L. Samuelis and Cs. Szabó

(Košice, Slovakia)

Abstract. The paper deals briefly with the technical background and the pedagogical issues of a specific implementation for the collection, assessment and archiving of the students' assignments written in Java. The implemented system automatically applies object-oriented metrics on the collected works in order to measure the characteristic features of the assignments. We interpret the measured values within a real Java course held in the 3rd term of the informatics bachelor study programme at the technical university. We observed that measuring the complexity of students' assignments is a valuable help for tutors from two points of view. It served data, which enable tutors to detect plagiarism and speeds up the selection process of oustanding works. On the other hand, the experiment supported also didactical goals, e.g. the awareness of the software measurement issues among students. This side effect seems to be a very valuable stepping stone towards understanding the more advanced software engineering topics in the following semesters.

1. Introduction

Teaching software engineering topics through open source-code case studies is not a novel approach in general. In spite of this fact only recently are offered sophisticated software engineering courses based on this approach. This

[1]This work was supported by VEGA grant No. 1/2176/05, and VEGA grant No. 1/0350/08 Knowledge-Based Software Life Cycle and Architectures

approach is important because industry demands mostly modifications of the existing programs and programs are less frequently built from scratch. Students, who study in this way, could gain complex skills in cognitive processes such as understanding, modification and reuse of the existing software. As much as 45% of resources are devoted to testing and simulation [1].

Earlier we have taught Java for three academic years following various books which were devoted to the Java introductory course. These courses (also the course offered by the Sun company, [3]) offer small separate illustrations how to implement certain constructs in Java and did not motivate smart students to progress on their own pace.

The other way, which we followed, is to assign independent projects, which cover a scope of lessons the course provides. This approach is closer to the above mentioned industrial practice. This variant offers students wider area for experimentation with the Java code. If we choose this possibility, it is very good to have a system that guides students through learning and then stores students' projects in a database for the evaluation and later reuse. That is why we implemented a system which should support these needs. The elaboration of robust courses based on the incremental analysis, the implementation of adequately selected set of open source-code case studies, and the management of students' deliverables for later reuse, require special attention and effort.

We show that a well-chosen case study facilitates the introduction of fundamental concepts in a coherent sequence. The basic idea is to guide students through explanations, models and pieces of code on the way to understand a complex code and application, so that they can apply the acquired knowledge in understanding further codes of similar complexity. The results fall broadly into two categories, they are technical and didactical in nature:

1. Technical results
   - Application that guides students through case studies by stepwise refinement. In other words, the developed application provides students and tutors with basic functionalities of a classical Learning Management System (LMS). These functionalities are e.g. loging and tracking of the student's progress. The *'pilot'* case study, that introduces students into Java, is devoted to the simulation of the *Automated Teller Machine* (ATM).
   - Application for archiving purposes and later measurement and evaluation of selected students' projects. After finishing the course (1 semester long) we collected the projects and measured the object-oriented characteristics of the projects. The application automatically measures the selected object-oriented features of the submitted projects.

2. Didactical results
   - We were especially interested in the selection of the outstanding projects, which should serve for learning purposes in the following academic years. The evaluated data helped during the marking process of the submitted projects and at the detection of the plagiarism. We show some snaps of the student's and the tutor's interface during the communication with the system and the evaluated data.
   - The afterwards discussions with students revealed additional unexpected side effect, which can be briefly described as increase of the students' awareness in the measurement aspects of software engineering.

The organization of the rest of the paper is as follows: in Section 2 outlines the motivation and the didactic aspects of the experiment; Section 3 offers an insight on the experiment. Section 4 is devoted to the analysis of the applied metrics and the final Section 5 contains summary and outlines the direction for future extensions of the experiment.

2. Motivation and didactic aspects

The idea of using the computer in the learning process is almost as old as computers are. Computer is being utilized almost in every phase of teaching and learning. Focusing on the role of the computer in the assessment process, we coincide with the standpoint expressed in the work of Bõszõrményi, L. [11, p.33], that *"Knowledge cannot be quantified and cannot be measured"*. This statement seems to be in contradiction with the title of the paper. But this is not the case. It warns us to be aware during the interpretation of the obtained results. It means, that we cannot measure the quality or the quantity of the student's knowledge based purely on the measured data. E.g., from the fact that a solution is more complex than another, we cannot deduce that a certain student is smarter than another. Tutor is the final judge of the running applications, evaluates the complexity, reviews, analyzes the documentation, and marks them.

The analysis of object-oriented programs may involve many features and we constrained the analysis only to several selected points. Quality metric results can reveal students' concepts, which they had to learn from theory and practice. Results of these measurements provide information how to improve the case studies and the selection of outstanding projects. We stress that the ultimate goal of the evaluation is the selection of outstanding applications, which may serve for the learning purposes for students in the following academic years.

Another motivation for doing this experiment was the observation that without archiving outstanding works we waste the effort and do not capitalize from previous solutions. This means that we wanted to reuse and to show the students' outstanding results for the coming students in the following academic years.

Last but not least, our department has established a pilot course devoted to the theory and practice of creating reusable e-learning courses based on the SCORM standard [8], [9]. We plan to utilize the available skills and to package the offered course according to the SCORM standard.

The experience and the discussions after the experiment revealed us a very important important didactic side effect, which will be discussed at the end in detail.

3. Brief description of the e-learning system

The system is implemented in Java using the Tomcat [4] technology and the SQL server. It provides Java case studies, which are divided into 13 lessons. The number of lessons is in accordance to the number of weeks in the semester. Lessons are weaved with small quizzes and students can verify their knowledge interactively. After completing all lessons, students' efforts culminate in creating their own ATM application. These applications are uploaded into the system at the end of the course. The system offers tools for the assessment of the submitted assignments. From the users' point of view, it provides GUI (Graphical User Interface) for 2 groups of users, namely for students and for tutors.

3.1. Student's interface of the LMS

At the beginning of the course students have to fill in a registration form for administrative purposes. After submitting this form they have to wait for the confirmation of the registration, which is provided by tutor. If student is successfully registered, s/he can log in into the system. After login the actual announcements are at disposal. These announcements are added and updated by tutor. The lessons are available after the registration. Figure 1. shows a piece of the text (some text is in slovak), which analyzes a specific class of the ATM simulation program.

These lessons support students with learning materials for object-oriented programming in Java step by step. The theoretical background of the case study is completed with practical examples, so students could obtain a complex knowledge of the problem. After successful completing all lessons, students should be

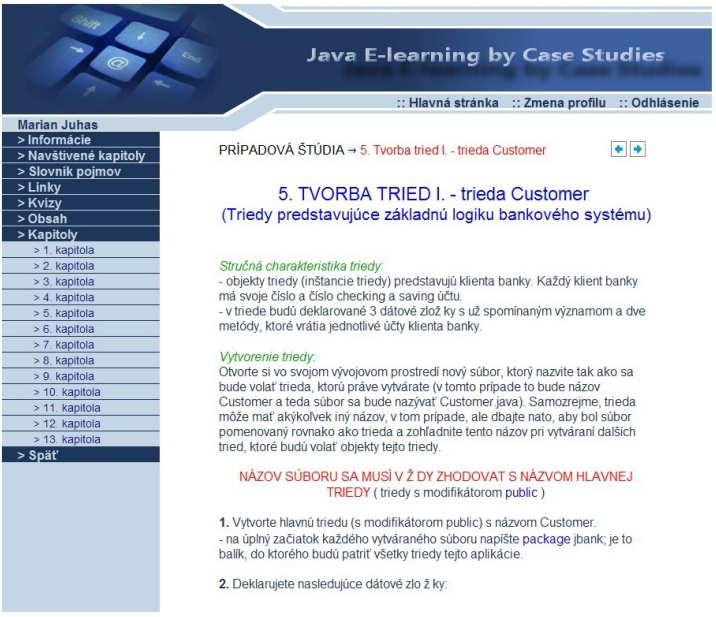

Figure 1. Student's interface of the LMS

able to create their own ATM simulation program, which was described in the practical part in the case study. The ultimate goal of this course is that students have to make their own ATM simulation program.

We have automated the submission process also for archivation purposes and for the evaluation of the object-oriented features of the submitted projects. Students submit the assignments as a JAR file with specific structure due to the automatic assessment of the projects. These specific instructions are available for students in detail throughout the course.

3.2. Tutor's interface of the LMS

As we mentioned before, tutor takes care about registering students into the system, updating and creating announcements. The most important issue for a tutor is to have an overview about the submitted projects in order to evaluate them for similarities and selection of the outstanding works. The LMS offers tools for grouping similar projects. This is one way how the tutor can check the originality of the projects. Every submitted project is evaluated by measuring several object-oriented features. Tutor then selects the outstanding projects in

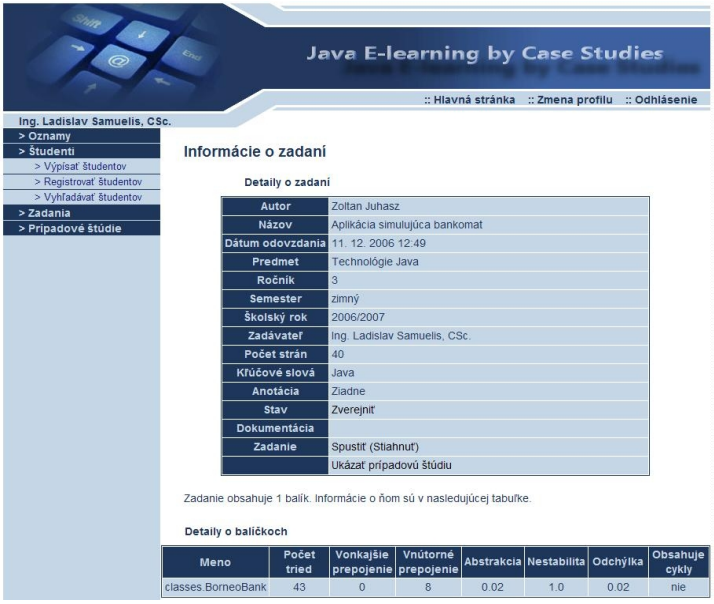

Figure 2. Tutor's view on the submitted assignment

order to put them into the pool of outstanding works for the reuse in the next academic year. Figure 2. shows the tutor's view on the submitted assignment.

3.3. Basic data of the experiment

The registered students already passed successfully programming in C language and basics of object-oriented programming. For the first time we tested the system with 104 registered students in academic year 2006/2007 in the summer semester of the 3rd term. During that period students had to understand and implement the project using Java. Full-time students created five study groups for consultancy purposes and there was one study group attended by part-time students.

As the system operated for the first time this year, the possibility that there will be problems with submitting assignments was very high. That's why the tutor and students from higher terms assisted continuously during the submission process.

As mentioned before, case study contains a tutorial how to create an ATM simulation program. The next graph in Figure 3. shows the number of students

with assignment similar to program described in case study tutorial. In other words, these data may provide some hints to detect plagiarism, but not this was the aim.

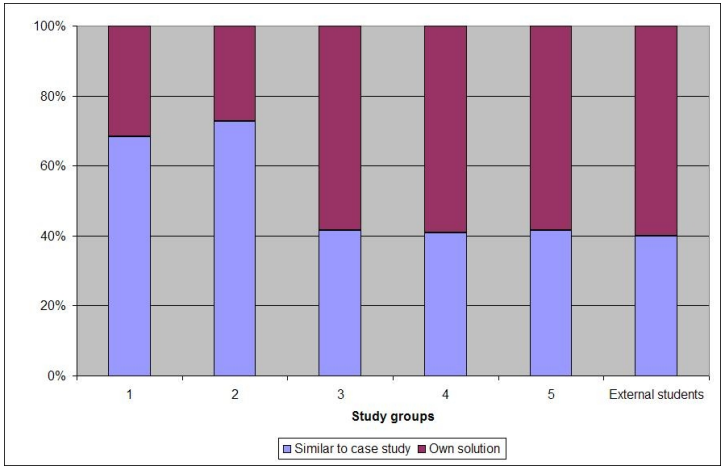

Figure 3. Similarity of the submitted projects

4. Object-oriented metrics applied for measuring the assignments

Software development process is no doubt a complicated one. The end product follows a chain of analysis, design, development and testing processes [10]. At each stage it is important to follow a well-defined methodology to ensure a quality end product. Software design and coding metrics play an important role in ensuring the desired quality.

4.1. Why metrics?

Object-oriented software is viewed as collection of objects. The functionality of the application is achieved by interaction among these objects in terms of messages. Whenever one object depends on another object, there is a relationship between those two objects (or classes on design level).

Object-oriented metrics can be a very helpful measuring technique to evaluate the design stability. There exist many clusterings, or abstractions, between the classes in the design phase. Inappropriate clustering directly influences the

vulnerability of the code during the maintenance process. More stable code is a prerequisite for easy maintenance. The results of object-oriented metrics may reveal dependencies, which cannot be observed at the first sight.

4.2. The JDepend tool and the measured object-oriented features

We have chosen JDepend [2] tool for the measurement because it is the best known tool between the programmers in practice. The outputs of the tool cover the basic features of Java packages and we considered this fact as sufficient argument for our experimental purposes. That is why we did not searched for more sophisticated tools.

After uploading the assignment into the system it is checked against the structure of the JAR file and then executed on a local PC. Next to this step the application is checked and evaluated against the predefined metrics.

JDepend traverses Java class file directories and generates design quality metrics for each Java package. This approach is automatic and provides interesting data for tutors. Tutor does the final assessment and marking in any case. Based of the JDepend package outputs, tutor has at disposal the following information about the quality of the source code:

4.2.1. Number of classes and interfaces.

The number of actual and abstract classes (and interfaces) in the package indicates the extensibility of the package.

4.2.2. Coupling.

It measures the interdependence of the classes. High coupling implies strong interconnections between classes, while loose coupling implies independence. Object-oriented designs reflect the real world as independent objects, which leads to loose coupling, if designed well. Inheritance causes tight coupling among classes. Because inheritance and polymorphism are heavily used in object-oriented paradigm, a study of metrics for object-oriented software becomes an important research project [6]. JDepend offers these kinds of couplings:

- **Afferent Couplings (C_a)** - The number of other packages that depend upon classes within the package is an indicator of the package's responsibility.
- **Efferent Couplings (C_e)** - The number of other packages that the classes in the package depend upon is an indicator of the package's independence.

4.2.3. Abstractness.

It measures the abstractness of a package by calculating the ratio of the number of abstract classes (and interfaces) in the package to the total number of classes in the package. Abstractness can be measured using the formula $A = N_a/(N_a + N_c)$, where:

- A represents a package's abstractness.
- N_a represents the number of abstract classes and interfaces in a package.
- N_c represents the number of concrete classes in a package.

An abstractness value 'zero' indicates a completely concrete package and the value 'one' indicates a completely abstract package.

4.2.4. Instability (I).

Stability is measured by calculating how easy it is to change a package without impacting other packages within the application. Let us examine how package dependencies impact stability.

Packages most heavily depended upon are the packages most heavily reused throughout an application. Packages with more incoming dependencies must, therefore, exhibit higher degrees of stability. In other words, packages most heavily depended upon are the most difficult to change. The number of incoming and outgoing dependencies is a major indicator in determining the stability or instability of a package. Packages containing many outgoing, but few incoming, dependencies are less stable, because the ramifications of change are less. On the other hand, packages containing more incoming dependencies are more stable, because they are more difficult to change. Stability can be calculated by comparing the incoming and outgoing package dependencies.

Instability is measured by calculating the ratio of efferent coupling to total coupling. The formula used to calculate instability follows: $I = C_e/(C_e + C_a)$ As I approaches 'zero', packages have many more incoming dependencies than outgoing dependencies, and the package is very stable. Stable packages are more difficult to change because the ramifications of changing a stable package may imply additional changes to many other packages. Packages with I approaching 'one' have many more outgoing dependencies than incoming dependencies, and the package is very unstable. Unstable packages are easier to change because few other packages in the application use them.

For any set of packages to compose an application, some packages must have incoming dependencies, while other packages have outgoing dependencies. Our goal in package design should not be that all packages are either completely stable or completely unstable. Instead, each package must be consciously made as

stable or unstable as possible. Those packages with many incoming dependencies must resist change, and exhibit higher degrees of stability. In object-oriented development, abstraction can be used to increase stability by separating what something does from how it does it. In Java, abstraction is manifested in abstract classes or interfaces. So more stable packages should be more abstract packages. Conversely, more unstable packages are more concrete packages.

4.2.5. Distance from the Main Sequence (D).

The perpendicular distance of a package from the idealized line $A + I = 1$. This metric is an indicator of the package's balance between abstractness and stability.

A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal packages are either completely abstract and stable ($x = 0, y = 1$) or completely concrete and unstable ($x = 1, y = 0$).

The range for this metric is 0 to 1, with D=0 indicating a package that is coincident with the main sequence and D=1 indicating a package that is as far from the main sequence as possible.

4.2.6. Package Dependency Cycles

Package dependency cycles are reported along with the hierarchical paths of packages participating in package dependency cycles.

4.3. Suggestions for extending metrics collection

Recently the metrics we provide is used to make the tutor's evaluative process easier and it is not useful for students at all. Our plan is to offer the advantages of software measurement for all users and it should be another teaching aid for students. This could improve the quality of the assignments. Pure numbers do not give any sense for students, hence it follows the brief explanation of metrics meaning has to be also provided. Our aim is to extend collection of metrics measured by system, namely Chidamber and Kemerer metric (henceforth, CK). Use of CK set of metrics and other complementary measures are gradually growing in industry. The object-oriented metrics proposed by Chidamber and Kemerer can be summarized as follows [5]:

- **Weighted Methods per Class (WMC)** - this is weighted sum of all the methods defined in a class.
- **Coupling Between Object Classes (CBO)** - it is a count of the number of other classes to which a given class is coupled and, hence, denotes the dependency of one class on other classes in the design.

- **Depth of the Inheritance Tree (DIT)** - it is a length of longest path from a given class to the root class in the inheritance hierarchy.
- **Number of Children (NOC)** - this is count of the number of immediate child classes that have inherited from a given class.
- **Response for a Class (RFC)** - this is count of the methods that can be potentially invoked in response to a message received by an object of a particular class.
- **Lack of Cohesion of Methods (LCOM)** - a count of the number of method-pairs whose similarity is zero minus the count of method pairs whose similarity is not zero.

These metrics should offer reinforcement that the software design is robust. While good metrics do not guarantee a quality design, good metrics do help bolster confidence. When used judiciously, these software metrics can be a valuable aid in assessing quality design [7].

5. Conclusions

In summary, students had at disposal within the course 4 different open-source case studies, which implemented an ATM simulation program. One of them was analyzed in detail. For the successful completion of the course every student had to create his/her own ATM simulation program. After finishing the course, we have got approximately further 8 outstanding projects available for learning in the next academic year.

More than 89% of students successfully submitted their assignments and more than 88% from these students submitted their projects before deadline. The most common problem during the submission process was the incompatibility of the JAR files. JAR files created in NetBeans or Eclipse IDEs were not accepted by the experimental system. This is the main reason why only about 46% of submitted assignments were accepted on the first attempt. This obstacle has to be resolved in the next version of the application. What concerns the similarity of the submitted works, we may conclude that approximately 50% of students tightly followed the code offered by case studies.

We observed also that most of students created just one package and put all classes into one package. This behaviour is typical for beginners. These results show that tutor has to focus on the explanation of importance of the packages.

The course was supported by the online access to the materials licensed from the Sun Microsystems, Inc. [3], Students had access both to the pool of source-

code case studies and to the Sun's materials. In this way students obtained blended learning.

The further planned extensions are:

- enhance statistic module of the LMS
- improving tutor's user interface
- enhance the robustness in order to accept variants of JAR files

To sum up the didactical results, we observed a very valuable side effect. In fact, it is a surprise for undergraduate students that the software complexity is somehow measured at all. This awareness influences the students' way of thinking. They are now asking: What does the software complexity mean? What is measured and why? These questions open new ways in students' minds and helps them to understand more advanced topics as concept location, software reusability, software comprehension, software architectures, etc., which are taught in the following semesters.

Showing and discussing the results with students opens them also historical aspects of software engineering, which is an iportant step from the didactic point of view. These observations coincide with the standpoint of Bõszõrményi, L. as expressed in [12].

References

[1] **Cartwright, M., & Shepperd, M.,** An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering,* 26(8)(2000), 786-796.

[2] **JDepend,** `http://clarkware.com/software/JDepend.html` (as of 21.3.2007)

[3] **Sun Learning Connection,** `https://learningconnection.sun.com` (as of 21.3.2007)

[4] **Apache Tomcat,** `http://tomcat.apache.org/` (as of 21.3.2007)

[5] **Subramanyam, R., Krishnan, M.S.,** Empirical analysis of CK metrics of object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering,* **29**(4)(2003).

[6] **Liu, X.,** *Object-oriented software metrics.* Department of Computer Science University of Manitoba Winnipeg, Manitoba, Canada, 1999.

[7] *Object-oriented design metrics,* `http://builder.com.com/5102-6386-5035294.html` (as of 14.4.2007)

[8] **Kollár, J., Samuelis, L. and Rajchmann, P.,** Notes on the experience of transforming distributed learning materials into SCORM standard specifications. *Information and Security*, vol. **14** (2004), 81-86, ISSN 1311-1493.

[9] **Bereczk, P., Samuelis, L.,** Implementation of IT related courses into SCORM standards. *New Information Technologies in Education for All: First International Conference Proceedings, 29-31, May 2006, Kiev*, Ukrainian National Academy of Sciences, 2006, 21-24.

[10] **Samuelis, L., Szabó, Cs.,** Notes on the role of the incrementality in software engineering. *Studia Universitatis Babes-Bolyai Informatica* **51**(2)(2006), 11-18.

[11] **Bõszõrményi, L.,** On methodic and didactics of the history of informatics, *MEDICHI 2007 - Methodics and didactic challenges of the history of informatics, Õsterrechische Computer Gesellschaft 2007, Bõszõrményi, L. (Ed.)*

[12] **Bõszõrményi, L.,** Teaching: people to people - About people (A plea for the historic and human view), *R.T. Mittermeir (Ed.): ISSEP 2005, LNCS* **3422**, 93-103, 2005.

M Juhás, Z. Juhász, L. Samuelis and Cs. Szabó
Department of Computers and Informatics,
Technical University of Košice,
Letná 9, 042 00 Košice, Slovakia
`juhasma@gmail.com, juhasz.zoli@gmail.com, ladislav.samuelis@tuke.sk, csaba.szabo@tuke.sk`