A PBL-Based, Integrated Learning Experience of Object-Oriented Programming, Data Structures and Software Design

Ayala L. Ribeiro
State University
of Feira de Santana
Feira de Santana, Bahia
Brazil 44036–900
ayalaedavi@gmail.com

Roberto A. Bittencourt
State University
of Feira de Santana
Feira de Santana, Bahia
Brazil 44036–900
roberto@uefs.br

***Abstract*—This innovative practice full paper presents an experience report of an approach integrating the teaching and learning of Object-Oriented Programming, Data Structures and Software Design in the second term of a Computer Engineering undergraduate program. Learning object-oriented programming requires acquiring high-level skills, which is negatively affected by present curricula and pedagogies. Isolated courses with teacher-centered approaches do not allow for the appropriate practice of such skills. For 15 years, our Computing Engineering undergraduate program has been pursuing an effort of curriculum integration and active learning practices based on problems and projects. This paper presents an experience report of our approach. This experience led us to important lessons learned with our approach: the acquisition of personal, interpersonal and technical skills provided by the approach; the benefits of knowledge integration through more authentic experiences and a more disciplined practice of software production; the need for careful problem planning; the main difficulties faced by instructors to manage the course; and the challenges faced by students to develop their skills.**

I. Introduction

Object-oriented programming (OOP) is central to software development in modern software industry. And learning it is hard since it requires more than mere concept understanding. Instead, it requires mastering high-level skills of analyzing, designing, implementing and validating software. Coupled with high-level skills, learners need to master language syntax, abstraction, data structures, integrated development environments and problem domain. To make matters worse, the object-oriented paradigm represents a different way of thinking, unfamiliar to apprentices that learned to code under a procedural imperative paradigm. Transitioning between paradigms leads to cognitive conflicts, which take time to solve [1].

Researchers have thoroughly researched the difficulties to learn programming. Robins describes those difficulties in terms of knowledge, strategies and mental models that work in three different levels of design, generating and evaluating programs [2]. Other researchers have focused on the advantages and disadvantages of particular paradigms and languages as the first to be learned [3]. However, there is still no consensus on the subject, as reference curricula show [4]. Kölling has thoroughly researched OOP as the initial paradigm for apprentices, leading to requirements for both OOP languages and environments for novices [5]–[8].

Regardless of language or paradigm choice, academic traditions in higher education typically maintain a learning order based on two initial programming courses, usually named CS1 and CS2, that leave advanced OOP issues to the CS2 course. In our opinion, part of the issues with the typical CS2 course are related to: i) the lack of proper space and time to practice high-level OOP skills, and ii) the use of teacher-centered learning approaches that do not let students acquire autonomy to practice those high-order skills.

There have been initiatives to promote more active, learner-centered approaches to teach OOP, such as problem-based learning (PBL) [9], peer learning [10] and flipped classroom [11], among others. Those initiatives are relevant, but, as we mentioned, are usually based on isolated courses inside a segmented curriculum, with different courses competing for student attention. This work tries to fill this gap, using a PBL approach together with an integrated curriculum. In the second term of our undergraduate program on Computer Engineering, students take an integrated course of object-oriented programming, data structures and software design aimed at acquiring and practicing high-level software development skills.

The goal of this work is to report our experience in the present integrated advanced programming course, based on a PBL teaching approach. It is based on 15 years of experience with PBL and an integrated curriculum.

The main learned lessons with our experience were: i) PBL allows acquiring both technical, personal and interpersonal skills; ii) the benefits of an authentic experience based on integration of knowledge; iii) the impact of problem planning on student learning; iv) the difficulties faced by instructors; and v) the challenges faced by students with the approach.

978-1-5386-1174-6/18/$31.00 ©2018 IEEE

II. Background

This section presents the essential background to better understand this experience report.

A. Learning Object-Oriented Programming

Robins et al. analyze the differences to learn programming using the object-oriented and the imperative paradigms [2]. They conclude by stating that novices have more difficulties to solve a long problem with the object-oriented paradigm compared to the imperative paradigm. They state that these difficulties are caused by a longer learning curve peculiar to the object-oriented paradigm itself.

Learning OOP as a second paradigm is thoroughly discussed in the literature. The difficulties may be associated with aspects such as the natural complexity of these languages or the complexity of professional development environments. Some defend starting with the imperative paradigm, following with the object-oriented paradigm [12]. Some authors argue that introducing paradigms such as object orientation in the CS1 course is not backed up by significant evidence of facilitating learning [13]. In contrast, Kölling (2003) suggests that object-oriented concepts must be taught from the beginning, arguing that the difficulties are in the tools used for teaching, not in the paradigm itself [7].

Kölling cites some important requirements for a programming language for novices [5]. It should be simple and easy to understand, purely object-oriented, secure, and high-level. It should allow an intuitive transfer of learned concepts to other languages. Some issues such as efficiency, often considered important for industrial programming languages, are of little value to a language of instruction. It is more important that the language can be supported by an appropriate teaching environment. In addition, there must be a proper development environment, allowing programmers to focus on what is really important: programming itself. Researchers have developed environments to facilitate learning of OOP, to minimize the difficulties found and, thus, to increase the engagement level. Tools such as Greenfoot [8], Alice [14] and BlueJ [7] are examples of environments that promote a playful experience, making programming fun and simpler for students.

B. Active Learning and PBL

Active learning is defined as any educational method that involves students in the learning process [15]. Prince discusses three popular approaches to active learning: collaborative learning, cooperative learning, and problem-based learning.

Problem-Based Learning (PBL) is an active learning approach developed by the physician Howard Barrows. It had its genesis between the late 1960s and early 1970s at McMaster University, Canada [16]. Initially idealized for medical education, the approach has gained acceptance and is increasingly present within a variety of disciplines in higher education. In PBL, students are responsible for building their own learning in a process of problem solving. Instructors act as a facilitators and are responsible for choosing problems to be solved. Students are autonomous to analyze and choose paths towards solving the problem [17].

In PBL, students are encouraged to find solutions by themselves. They are in charge of searching information sources for learning. The instructor prepares the environment, helps students to relate to the problem, arranges a work structure, addresses issues between students, re-equates the problem, facilitates product or performance production and encourages self-assessment [17]. He or she acts as a facilitator, posing questions, so that students are in charge of the process of knowledge creation. This process generates learning for both students and the instructor [18].

With PBL, not only are students constantly encouraged to learn, but they also play an active role in the process of building knowledge [19]. A problem in PBL is a trigger to motivate study. Generally, it follows a learning cycle, repeated over the duration of the problem: 1) Students are presented with a problem before any preparation or study; 2) Students come together in groups to organize ideas and recall previous knowledge related to the problem; 3) Through discussion, students pose questions known as learning issues that deal with aspects of the problem that they do not understand; 4) Learning issues are ranked in order of importance, and students set learning goals for independent and group study [19], [20].

PBL values deep understanding more than rote learning, even though the latter is important for learning. The greater the understanding of a particular subject, the easier it is to memorize and, consequently, to learn. However, learning that is only at the level of memorizing has little value for both social and professional lives. This is one of the main issues of lectures that emphasize content only. With PBL, it is different because this approach allows students to observe and analyze attitudes and values that are not present in the typical teacher-centered approach. As problems are presented in a real context, they favor transfer of both knowledge and skills learned in class to professional lives [17], [21]. PBL allows students to adapt to changes, encourages critical thinking, teaches learning how to learn as well as to work as a team [22].

C. PBL in Computing Education

O'Grady carried out a systematic review to investigate how PBL has been used in computing curricula [23]. The study found 63 papers in computer science, computer engineering, information systems, information technology and software engineering undergraduate programs. In these programs, a wide variety of courses have used PBL as teaching approach: software engineering, computer programming, software quality and operating systems, among others. They conclude that PBL penetration in computing curricula is still superficial. In many cases, faculty members are working in isolation and are faced with the challenge of introducing PBL into curricula, which remain essentially instructionalist. In some cases, PBL is a last resorted measure to improve either retention or performance.

In Brazil, PBL experiences in computing are still very restricted. In the literature, PBL has been used in six courses

in the Software Engineering program at the Federal University of Pampa (UNIPAMPA). These courses integrate different contents within problem scenarios similar to professional practice in an interdisciplinary and cross-cutting way [24]. Another experience that has integrated the PBL approach to the curriculum is the Computer Engineering program at the State University of Feira de Santana (UEFS) [25].

D. Related Work

Vihavainen et al. describe an evaluation of an active learning approach for teaching introductory programming [10]. This course lasted one semester and used the Java programming language. The approach was split into two units: introductory programming, where they introduced basic OOP concepts, and advanced programming, where knowledge learned in the initial phase was deepened. The approach reduced the number of lectures from five hours a week to only two hours, so that they could invest in classes focused on practical activities of incrementally growing difficulty. In addition, there was constant support from experienced instructors. Authors conclude that the use of their extreme learning method, combining assistance by experienced instructors with a set of values and practices yields positive results. The results pointed to reduced dropout and failure rates as well as an increase in course acceptance.

Herala et al. report the use of an active learning, flipped classroom approach in an object-oriented programming course [11]. This approach proposes the full inversion of the teaching model, with less lectures, and more productive and participating discussions. In addition to the teaching approach, they also changed the programming language, learning materials, weekly exercises, programming projects and assessment. They switched the programming language from C++ to Java, and assessed students through online exams instead of written exams. To support the flipped classroom approach, the course had several mini-lecture videos, each lasting from 15 to 30 minutes. Authors conclude by stating that the flipped classroom approach is suitable for teaching object-oriented programming.

Active problem-based learning (PBL) approaches to teach object-oriented programming (OOP) have been used in some universities. Kay et al. report the evaluation of a pioneering PBL approach where they redesigned the syllabus of a CS1 course, moving from an imperative programming language (Pascal) to an object-oriented one (Java), facilitated by the BlueJ environment [9]. Authors conclude by summarizing their three-year experience with some achievements. Students acquired OOP skills, independence, autonomy, ability to work in groups, and developed critical thinking, problem-solving and communication skills, among other relevant soft skills.

An experience of integrating object-oriented programming, data structures and systems design is reported in an integrated course based on a problem- and project-based learning approach. The experience lasted two academic terms, and the main lessons learned were: integration of knowledge enables more authentic experiences and practices of more disciplined software production; the PBL approach allows to acquire broader communication, teamwork and self-directed learning skills; Software maintenance problems should be avoided for they can reduce motivation when accumulated deficiencies arise; the dosage of new concepts in the proposed problems must respect a gradual process and students' assimilation capacities [26].

Ferreira et al. (2007) describe an active project-based learning approach in an introductory laboratory course on object-oriented programming and their experiences as instructors. In this course, students must implement a small to medium-sized interactive computer game in one semester using the Java language. Authors conclude by stating that students performed well with this approach, and that learning based on a practical project enriches students' interactions with interdisciplinary content from the theoretical course, and it helps them realize the relevance of a practical course to real life issues [27].

III. Methodology

We use the experience report as our methodological strategy. In this section, we describe the educational scenario, study participants, planning of the theoretical modules and the integrator module, and student assessment.

A. Scenario

The State University of Feira de Santana (UEFS) offers the Computer Engineering program since 2003. This program uses the PBL approach as its main teaching and learning strategy, and is based on a flexible curriculum, favoring constant updating of course content [28]. The program is also characterized by the integration and interdependence between curricular components that groups courses with related contents in a same academic period, particularly evidenced by the curricular components named Integrated Studies.

The Integrated Study (IS) is an integrating component focused on a given theme and is organized into modules. During the integrated study, students are introduced to given comprehensive problems. To solve problems, they need to acquire new knowledge, which is grouped into modules. Presently, nine integrated studies are offered during the program.

Programming concepts are present in two integrated studies in this program. The *Algorithms IS* is a curricular component offered in the first semester, integrating introductory ideas of algorithms, basic data structures (arrays and registers) and structured programming in an imperative language. In the second term, the *Programming IS* integrates object-oriented programming, advanced algorithms and data structures, and software design. Activities build on the experience gained in the Algorithms IS.

B. Participants

Our study took place in the second half of 2017 with students from the *Programming IS*. Classes were heterogeneous, composed by 46 freshman and sophomore students: 44 male and 2 female. All of them had previous experience with programming since they had passed the *Algorithms IS*.

The theoretical modules were taught by two instructors: one instructor led the Software Design module and a second instructor led both the Data Structures and the Object-Oriented Programming modules. Not all students who take the theoretical module take the integrator module, and vice-versa, because of previous dropouts or failures. However, when taking the IS for the first time, a student is registered in all component modules. The integrator module (IM) had 25 students, distributed in four tutorial groups, each composed by at least five and at most ten students, with one tutor for each tutorial group. Tutors can either be faculty members or teaching assistants.

C. Planning

The general goal of the *Programming IS* is that students be capable of designing, developing and testing object-oriented software, using appropriate algorithms and data structures, mastering the concepts and fundamentals underlying the methods, techniques and tools used. Specific goals are derived from the general goal and from the desired skills (e.g., being able to write, compile and debug object-oriented programs in Java; being able to choose and implement data structures appropriate to a search problem).

The *Programming IS* consists of three theoretical modules and the integrator module (IM). The theoretical modules entail 30 hours each and are composed by three modules: Object-Oriented Programming, Data Structures and Software Design. In these modules, classes are typically in the form of lectures. They happen in ordinary classrooms, and instructors use whiteboard, projector and computer as support materials. The *Programming IM* happens in parallel, with two weekly sessions of PBL tutorials with two hours each, and a total workload of 60 hours. Sessions take place in tutorial rooms where small groups of students gather, aided by a tutor. In these sessions, students use the whiteboard and their laptop computers. Source code in Java is typically produced in IDEs such as Eclipse or NetBeans.

Tables I, II and III respectively present the course plan for the theoretical modules of Object-Oriented Programming, Data Structures and Software Design. Theoretical modules are divided into three units. The order of theoretical concepts in each unit is usually planned to follow the order of acquisition of the needed concepts in the solution of the proposed problems, although there is no strict correspondence between them.

The course plan for the *Programming IM* follows a format where problems are drawn from a real world scenario. Problems follow a spiral of increasing complexity. Four problems were proposed and distributed over a semester. For each problem, students need to learn new concepts and acquire practical programming skills, as shown in Table IV.

D. Student Assessment

The second academic term of 2017 lasted between four and five months. Each theoretical module had three written exams, which correspond to the grades of Units I, II and III. Written exams had both multiple-choice and essay questions. Grades

Table I
COURSE PLAN FOR THE OBJECT-ORIENTED PROGRAMMING THEORETICAL MODULE

Unit	Skills	Concepts
I	Understanding object-oriented modeling and programming, and writing basic object-oriented programs.	1. Programming Paradigms 2. Object-oriented programming 3. Objects and Classes 4. Fields and State 5. Methods, Messages and Behavior
II	Being able to write, compile, debug and test programs in Java programming language. Applying object-oriented data structures for modeling simple and complex data from real world problems.	6. Encapsulation 7. Inheritance 8. Polymorphism and Dynamic Binding 9. Relationships and Composition 10. Coupling and Cohesion 11. Class and Object Implementation 12. Final Classes, Fields and Methods
III	Handling errors and exceptions in programs in Java. Creating simple graphical user interfaces in Java.	13. Abstract Classes and Interfaces 14. Static Attributes and Methods 15. Handling Exceptions 16. Basic Data Structures (Collections) 17. Graphical user interfaces and event-oriented programming 18. File Handling

Table II
COURSE PLAN FOR THE DATA STRUCTURES THEORETICAL MODULE

Unit	Skills	Concepts
I	Knowing how to compute complexity of basic algorithms. Designing basic data structures such as lists, queues and stacks	1. Complexity of algorithms 2. Memory allocation 3. Arrays (tables) 4. Simple and linked lists 5. Queues 6. Stacks 7. Linear search
II	Knowing the main search and sorting algorithms	8. Binary Search 9. Sorting algorithms (HeapSort, BubbleSort, SelectionSort, MergeSort, InsertionSort, QuickSort) 10. Binary trees in arrays 11. Heaps 12. Priority queues 13. Bucket Sort
III	Designing and implementing advanced data structures like trees and hash tables. Knowing how graphs work, their representation and and developing search algorithms in directed and undirected graphs.	14. Binary Trees 15. Balanced Binary Trees (AVL) 16. B Trees 17. Hashing 18. Hash Tables 19. Graph representation 20. Graph algorithms 21. Shortest path algorithms

Table III
COURSE PLAN FOR THE SOFTWARE DESIGN THEORETICAL MODULE

Unit	Skills	Concepts
I	Knowing, understanding and applying methods, techniques and tools to design OO software.	1. UML: Class diagram 2. Domain models: (concepts, associations and attributes 3. Building domain models 4. Design class diagrams 5. Operations 6. Association Attributes 7. Collections 8. Relationship Navigability 9. Design of Relationships 10. Mapping design to source code
II	Producing documentation for designs based on essential diagrams of the Unified Modeling Language (UML) Knowing and using software requirements artifacts, conceptual analysis, source code and tests in order to design software systems with an integrated view of the software life cycle.	11. Unit Tests 12. JUnit Framework 13. Generalization and Specialization 14. Supertypes and subtypes 15. Polymorphism 16. Abstract types: abstract classes and interfaces 17. Polymorphism with abstract types 18. Multiple inheritance 19. System tests 20. Acceptance tests
III	Choosing general patterns to assign responsibilities and design patterns to solve problems of OO software design	21. Interaction diagrams in UML 22. Communication Diagrams 23. General responsability assignment software patterns (GRASP) 24. Observer design pattern 25. MVC design pattern

vary between zero and ten and the arithmetic average of the exams is the final module grade. Students pass with an average grade of seven or higher.

In the *Programming IM*, both student performance in tutorial sessions and delivered student products account for the grade of each proposed problem. Performance is measured by attendance, punctuality, participation and effective contribution. A student product is usually a software system and, at times, its Javadoc documentation and a written technical report. Products are assessed through assessment criteria given to students during the course of the problem, and take into account the knowledge and use of software design, object-oriented programming and data structures concepts and skills. Product assessment may also include a written report, with particular assessment criteria for technical writing. The product delivered accounts for 70% of the problem grade and student performance accounts for the other 30%. Final IM grade is the average of the problem grades. Students pass with an average grade of seven or higher.

E. Data Collection and Analysis

We followed a qualitative protocol based on observations and semi-structured interviews to better understand our experience and derive learned lessons. The first author of this paper acted as a non-participant observer that observed both instructors and students in class, both in the theoretical modules and in the IM. After the course, we interviewed two instructors and three students, using different semi-structured interview guides for instructors and students. We analyzed the textual data using an open coding protocol based on content analysis to derive findings directly from the data.

In this paper, to respect anonymity, instructor interviews are identified with the letter I and a number, while student interviews are identified with the letter S and a number.

IV. OUR EXPERIENCE

Here we present the results of our experience, using the proposed problems as our narrative thread.

The tutorial sessions follow established rules, obeying the dynamics of the PBL approach. At the beginning of the session, students choose among them a session coordinator, a board scribe and a reporter. The session coordinator must lead the group, foster participation, keep the appropriate dynamics, and direct discussions on possible problem solutions. The reporter takes notes on the issues discussed, and orders ideas in a session report. The board scribe takes notes on the whiteboard, expressing ideas, facts, questions and learning goals raised by the students. In the first problem session, a problem is presented to students, who read and interpret the scenario together, and raise relevant facts and ideas to solve it. After this phase, they raise learning issues that may lead to solving the problem. Finally, they establish an action plan to solve the issues raised, deriving explicit learning goals. At the end of the tutorial session, the reporter generates the session report and shares it with the group.

Problem 1 was presented to students in the first class of the *Programming IM*. In this problem, students designed and implemented a system to schedule electoral bio-metrics registration. The problem presented the scenario with a narrative, user stories and a conceptual model. Students had to design and develop a solution in three weeks. To solve it, they needed to acquire introductory concepts of classes and objects, linked lists, and the Iterator and MVC design patterns.

We observed that despite the large number of new concepts in Problem 1, most students were able to deliver the product in a timely manner. Understanding the conceptual model and class diagram seemed relatively simple. However, designing the system from these documents proved more difficult. Students learned concepts in the theoretical modules concurrently with problem discussion in the tutorial sessions, which aided to clear students' doubts. However, students frequently needed to seek additional materials to achieve learning.

Problem 2 was given to students after Problem 1's deadline. In this new problem, students developed a tool for electronic auction. From the problem scenario, user stories, given functional tests and a class diagram, students designed and implemented a system. To do so, they needed to use new concepts of data structures, software design and object-oriented programming, such as inheritance, priority queues, sorting algorithms and the JUnit framework.

Table IV
COURSE PLAN FOR THE PROGRAMMING INTEGRATOR MODULE (IM)

Problem	Theme	Skills	Concepts
01	Scheduling electoral Bio-metric Registration	Designing and implementing a simple object-oriented program from given software requirements and analysis.	1. Classes and objects 2. Fields, methods, and constructors 3. Console Input and output in Java 4. Interfaces in Java 5. Basic MVC design pattern 6. Domain model 7. Design Class Diagram 8. Abstract data types in data structures 9. Linked Lists: Implementing Operations 10. Iterator design pattern: implementation and use.
02	Electronic Auction Tool	Designing, implementing and testing a small software system based on basic code reuse, basic data structures and disciplined software design from given software requirements and analysis.	1. Constructors 2. Method overload 3. Object Composition 4. Simple inheritance 5. MVC design pattern 6. Domain model 7. Design class diagram 8. Unit Tests 9. Stacks, Queues, Priority Queues 10. Sorting Algorithms
03	Stock Portfolio Manager	Designing, implementing and testing a software system based on advanced code reuse techniques, advanced data structures and disciplined software design from given software requirements and analysis.	1. Text files 2. Binary files and serialization 3. Handling Exceptions 4. Façade design pattern 5. MVC design pattern 6. Unit Tests 7. System and Acceptance tests 8. Design Class Diagram 9. Binary trees: representation and algorithms 10. Balancing Binary Trees 11. Documentation with Javadoc
04	Travel Management App	Analyzing, designing, implementing and testing a GUI-based software system from advanced code reuse techniques, advanced data structures and disciplined software design.	1. Graphical user interface components in Java 2. Handling interface events in Java 3. Collections from the standard Java library 4. Abstract classes and polymorphic inheritance 5. MVC design pattern 6. Unit Tests 7. System and Acceptance tests 8. Design Class Diagram 9. Hash Tables: Representation and Algorithms 10. Graphs: representation and traversal 11. Shortest path algorithm for graphs

Problems are designed with increasing complexity. In the second problem, students had to write their code to pass given functional tests, build a design class diagram, implement and use priority queues, choose a sorting algorithm and, ultimately, write their product report. We noticed that students had difficulties to understand the requirements and to elaborate the design class diagram. They needed tutor advice at times, especially to redirect discussions to avoid moving away from the proposed problem core.

We observed difficulties to choose the sorting algorithm and to pass functional tests. On the other hand, we perceived that students grasped the relevance of the concepts worked, especially the concept of inheritance and the possibility of code reuse. Passing functional tests also gave them more confidence, by allowing systematic flaw identification in the product under development. Most students delivered the project within the stipulated four-week deadline.

In Problem 3, students designed and implemented a stock portfolio manager. This problem only contained informal specifications in a narrative, besides user stories. Students learned concepts of balanced binary trees, file reading and writing, Javadoc documentation, interfaces and the *Façade* design pattern. Unlike previous problems, students had to implement all unit tests for the classes used, except for the *Façade*. All classes and tests had to be documented using *Javadoc*. In addition to source code, students also had to deliver a design class diagram for the system.

In this problem, we observed that students had difficulties to interpret the problem, which we blame to the lack of familiarity with the problem domain. Lack of knowledge about the stock market and stock exchange were the main reasons identified, in addition to difficulties related to technical concepts. Most students had difficulty implementing the balancing of an AVL binary search tree. Nevertheless, the greatest

difficulty was to understand and use the *Façade* design pattern.

Most students either did not finish Problem 3 in a timely manner or delivered an incomplete product. One of the possible explanations is the high complexity and the unknown domain.

Finally, with Problem 4, students designed and implemented a travel management application. Unlike previous problems, it only presented a problem scenario and the informal specifications from the scenario, but no user stories. To solve it, students needed to develop a graphical user interface (GUI), and learn the concepts of hash tables and graphs. In addition to implementation, they had to provide unit tests for all classes, system tests and documentation in Javadoc. They also had to write their own user stories and deliver a design class diagram for the system. This information had to be included in a technical report, in addition to the source code.

On this last problem, students had the best results. They did not show difficulties with concepts of OOP or data structures. We noticed particular issues with software design. Students had difficulties to build the design class diagram, especially associations and class navigability. On the other hand, Problem 4 proved to be a stimulating experience for students. One of the possible explanations is the use of a GUI, allowing immediate visual feedback, and the chosen domain, since the problem dealt with an application similar to existing apps for travel planning.

V. Learned Lessons

The main learned lessons of our experience were: acquisition of personal, interpersonal and technical skills; benefits of knowledge integration; need for careful planning of the problems; and the difficulties faced by instructors and students.

The PBL approach characterizes a learning strategy, whereby students are confronted with contextualized and ill-structured problems for which they strive to find meaningful solutions. The path towards solutions allows **acquisition of personal, interpersonal and technical skills**. Personal skills are related to autonomy, ability to solve problems and to make decisions. Interpersonal skills involve abilities to work in groups, leadership and communication. *"What is best in the PBL method for students is their knowledge construction, they try to learn on their own, incentive, creativity, autonomy, development of interpersonal skills, which do not happen in the traditional approach. Communication is fully stimulated in the PBL approach"* (I1). Technical skills are related to the very dynamics that the PBL approach advocates. The experience allows the improvement of technical skills acquired with the practical experience provided during problem solving. *"The PBL method is good because it makes us learn in practice."* (S1).

The benefits generated by **knowledge integration** are one of the main contributions of our approach to learn software development. Students experienced object-oriented programming, data structures, and software design concepts and skills in a pedagogical framework where theory and practice walk together. *"The IM helps with practice. In the IM, they have the opportunity to practice, to code, to write, to test recursion, to see a larger real problem. I can feel that they are being stimulated in the IM, which consequently leads to learning in the theoretical course."* (I2). This allowed students to undertake more authentic projects by gaining experience in the various stages of the software life cycle. They also learned a more disciplined way of working, which will be useful in their following courses and professional lives. *"The PBL method is challenging, and it is closer to real life. It seems to detach from something strictly academic, approaching very much the real world."* (S2).

Careful problem planning has a relevant impact on learning. A well-crafted problem can raise motivational levels, while a poorly planned problem can have catastrophic outcomes. The quality of PBL problems is among the biggest challenges of the approach. Excess of concepts, problem domain and the required complexity are issues that must be considered. Concepts in the proposed problems must follow a gradual learning process, so that students can better assimilate or accommodate them. The choice of problem domain should take into account student reality, and it is a strategy to make problems more attractive and motivating. Problem complexity must ensure that students can solve the problem with cooperation. *"This is what I see... it is better when you're able to design a problem closer to students' experiences. Then, for instance, the smartphone travel app that simulates touristic routes was the most well received by students."* (I1).

The difficulties faced by instructors are related to different issues. One of them is when to intervene in tutorial sessions. Another is how to appropriately handle issues with interpersonal relationships. Student assessment is yet another. Instructors play the role of facilitators and are responsible for guiding and monitoring group work. They must have the capacity to play different roles in the various stages of the process. At first, they should be able to realize how a person behaves when he or she has limited knowledge of a given content. Then, they must adopt strategies where their collaboration is relevant to the process. On the other hand, their interventions can not be exaggerated, otherwise they would invade students' roles as advocated by PBL proponents. Appropriate intervention requires empirical skills, and instructors must be sensible to the process of knowledge development. In addition, instructors must know how to deal with interpersonal relationships. They must know how to manage conflicts, be flexible, check inconsistencies, consider alternatives, and always lead the group to greater productivity and cohesion.

Assessment in PBL is also a difficult task. Some of the main difficulties are: assessing student performance, checking the authenticity of student products and setting up appropriate assessment criteria. Student performance is measured by their participation in tutorial sessions. Student participation in the sessions is extremely important to exchange ideas and develop hypotheses for problem solution. Nonetheless, students have a diversity of psychological profiles. Various are shy, introverted and silent while others are participant, extroverted and communicative. One can not say that the quiet or shy

students are not studying or participating. Sometimes a short, yet precise, contribution can be significant. Understanding the diverse subjectivity of student profiles and finding a balance during assessment are some of the greatest challenges for the instructor. *"In the PBL method, students that have a hard time to communicate, sometimes introverted, shy and the like... they have an opportunity to develop important communication skills... to at least improve those important communication qualities, in different ways, learning how to participate in a meeting, expose ideas and opinions."* (I1).

Ensuring authenticity of student products is another issue encountered by instructors. This is typical of work performed outside class. Ensuring that there is no plagiarism is not an easy task. Individual technical reports try to cope with this issue, but they are not always enough. Moreover, when assessing products, instructors need to develop assessment criteria. In order to assemble criteria, instructors need feedback from the other tutors, and this is not always as simple as it seems. The greatest difficulty lies in finding the right balance of assessment criteria for each problem.

Students face difficulties during the transition between programming paradigms, both with their abstraction skills and in developing their autonomy. The transition from the imperative to the object-oriented paradigm raises a cognitive conflict in students, often time-consuming. On the other hand, learning object-oriented programming is not easy. The difficulties may be caused by the complexity of the concepts to be learned in a short period of time, by the intrinsic complexity of these languages and also by the professional development environments. *"In the beginning, there is the shock of changing from an imperative language to an object-oriented language, but when you learn, you notice that it is way better"*. (S1). Finally, the PBL approach requires development of student autonomy. Those who are not self-directed learners face difficulties in this new context. *"The Programming IM has a very large strength to involve students, but not all of them. Students that have difficulties with programming, with the profile of our program, that have personality issues, maybe do not remain much motivated."* (I1)

VI. Conclusions

This work presented an experience of integrating the courses of object-oriented programming, data structures and software design into an integrated study of a Computer Engineering undergraduate program. The experience happened in the second half of 2017, and used a problem-based learning (PBL) approach. This report presented the scenario, participants, planning and assessment, as well as the main results of the experience and important lessons learned.

Results are described as a temporal narrative of the problem scenarios and the tutorial sessions. We describe the theoretical concepts needed and how they were used in the integrator module (IM) for each of the four proposed problems. We also highlight students' difficulties in the different steps of the tutorial sessions.

We realized that the path taken by students to solve problems allows the acquisition of personal, interpersonal and technical skills. Among them, we highlight autonomy, problem solving, decision making, working in groups, leadership, communication, and technical programming skills. Furthermore, the designed integration of knowledge enables more authentic experiences and more disciplined software production practices. We also realized that problem planning directly interferes with student learning, and that a poorly crafted problem may lower motivational levels.

Instructors face difficulties to assess student performance because of its subjectivity and the need to take into account students' psychological profiles. They also have issues with plagiarism and to devise assessment criteria. Intervention during tutorial sessions requires empirical skills, and sensibility to the process of knowledge development. In addition, instructors must know how to deal with interpersonal issues. Students also face difficulties, ranging from the transition between programming paradigms to the autonomy required by the PBL approach.

In future work, we plan to evaluate the data collected in this integrated course both qualitative and quantitatively to provide further insights on the PBL approach in our scenario. We also intend to measure motivation levels in the theoretical modules of Data Structures, Object-Oriented Programming and software Design, and in the Programming integrator module that make up this integrated study.

Acknowledgment

The authors would like to thank both the students and faculty members of the State University of Feira de Santana (UEFS) that took part in this experience.

References

[1] T. Jenkins, "On the difficulty of learning to program," in *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, vol. 4, no. 2002, 2002, pp. 53–58.

[2] 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.

[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," *ACM SIGCSE Bulletin*, vol. 39, no. 4, pp. 204–223, 2007.

[4] ACM/IEEE-CS Joint Task Force on Computing Curricula, "Computer science curricula 2013," ACM Press and IEEE Computer Society Press, Tech. Rep., December 2013.

[5] M. Kölling, "The problem of teaching object-oriented programming, Part 1: Languages," *Journal of Object-oriented programming*, vol. 11, no. 8, pp. 8–15, 1999.

[6] ——, "The Problem of Teaching Object-Oriented Programming, Part II: Environments," *Journal of Object-Oriented Programming*, vol. 11, no. 9, pp. 6–12, 1999.

[7] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ System and its Pedagogy," *Computer Science Education*, vol. 13, no. 4, pp. 1–12, 2003.

[8] M. Kölling, "The greenfoot programming environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 14, 2010.

[9] J. Kay, M. Barg, A. Fekete, T. Greening, O. Hollands, J. H. Kingston, and K. Crawford, "Problem-Based Learning for Foundation Computer Science Courses," *Computer Science Education*, vol. 10, no. 2, pp. 109–128, 2000.

[10] A. Vihavainen, M. Paksula, and M. Luukkainen, "Extreme apprenticeship method in teaching programming for beginners," in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 93–98.

[11] A. Herala, E. Vanhala, and U. Nikula, "Object-oriented programming course revisited," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, ser. Koli Calling '15. New York, NY, USA: ACM, 2015, pp. 23–32.

[12] R. A. Baeza-Yates, "Teaching algorithms," *ACM SIGACT News*, vol. 26, no. 4, pp. 51–59, 1995.

[13] P. J. Burton and R. E. Bruhn, "Teaching programming in the oop era," *ACM SIGCSE Bulletin*, vol. 35, no. 2, pp. 111–114, 2003.

[14] S. Cooper, "The design of alice," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 15, 2010.

[15] M. Prince, "Does Active Learning Work? A Review of the Research," *Journal of Engineering Education*, vol. 93, no. 3, pp. 223–231, 2004.

[16] H. Barrows and R. Tamblyn, *Problem based-learning: An approach to medical education*. Springer Publishing Company, 1980, vol. 1.

[17] R. Delisle, *How to use problem-based learning in the classroom*. Ascd, 1997.

[18] J. R. Savery and T. M. Duffy, "Problem based learning: An instructional model and its constructivist framework," *Educational technology*, vol. 35, no. 5, pp. 31–38, 1995.

[19] C. da Silva Cintra and R. A. Bittencourt, "Being a PBL teacher in Computer Engineering: An interpretative phenomenological analysis," in *2015 IEEE Frontiers in Education Conference (FIE)*, Oct 2015, pp. 1–8.

[20] D. M. B. Santos, G. R. P. R. Pinto, C. P. P. Sena, F. C. Bertoni, and R. A. Bittencourt, "Aplicação do Método de Aprendizagem Baseada em Problemas no Curso de Engenharia de Computação da Universidade Estadual de Feira de Santana," in *XXXV Congresso Brasileiro de Educação em Engenharia*, 2007.

[21] M. A. Albanese and S. Mitchell, "Problem-based learning: a review of literature on its outcomes and implementation issues." *Academic Medicine : Journal of the Association of American Medical Colleges*, vol. 68, no. 1, pp. 52–81, 1993.

[22] J. A. M. Ortiz, A. G. González, A. P. Marcos, M. Victoria, and A. Nardiz, "Aprendizaje basado en problemas: una alternativa al método tradicional," *Revista de Docencia Universitaria*, vol. 3, no. 2, 2003.

[23] M. J. O'Grady, "Practical problem-based learning in computing education," *Trans. Comput. Educ.*, vol. 12, no. 3, pp. 10:1–10:16, Jul. 2012.

[24] J. F. P. Cheiran, E. de M Rodrigues, E. L. de S Carvalho, and J. P. S. da Silva, "Problem-based learning to align theory and practice in software testing teaching," in *Proceedings of the 31st Brazilian Symposium on Software Engineering*, 2017.

[25] M. F. Angelo, A. C. Loula, F. C. Bertoni, and J. A. M. Santos, "Aplicação e avaliação do método PBL em um componente curricular integrado de programação de computadores," *Revista de Ensino de Engenharia*, vol. 33, no. 2, pp. 31–43, 2014.

[26] R. A. Bittencourt, C. A. Rodrigues, and D. S. S. Cruz, "Uma Experiência Integrada de Programação Orientada a Objetos, Estruturas de Dados e Projeto de Sistemas com PBL," in *XXXIII Congresso da SBC – XXI WEI.*, 2013.

[27] G. M. Ferreira, M. Z. Nascimento, K. D. R. Assis, and R. P. Ramos, "Teaching object oriented programming computer languages: Learning based on projects," in *Proceedings of the ICSEA*, 2007.

[28] R. A. Bittencourt and O. A. Figueiredo, "O Currículo do Curso de Engenharia de Computação da UEFS: Flexibilização e Integração Curricular," in *Anais do XXIII Congresso da SBC*. Campinas, São Paulo: SBC, 2003, pp. 171—-182.