Using PDCA as a General Framework for Teaching and Evaluating the Learning of Software Engineering Disciplines

Sérgio Mergen[1], Fábio N. Kepler[1], João Pablo S. da Silva[1], Márcia C. Cera[1]

[1]Universidade Federal do Pampa (UNIPAMPA) - Campus Alegrete
Av. Tiarajú, 810, Ibirapuitã, Alegrete/RS

{sergiomergen, kepler, joaosilva, marciacera}@unipampa.edu.br

Abstract. *Software engineering disciplines need to be taught in contexts as diverse as undergraduate courses and large corporations training programs. A primary challenge in teaching such disciplines, in any context, is to quickly and effectively evaluate the students learning and measure their strengths and weaknesses. Another challenge is to make students of different instances of a discipline end up with the same basic foundations, turning knowledge independent of the instructor. To overcome these challenges we propose an approach for software engineering teaching based on adapted PDCA cycles and checklists as instruments of evaluation. We also report a case study which shows the implementation of this approach in teaching a first year undergraduate software engineering course. With a careful definition of checklists, the use of the adapted version of PDCA as a methodology for software engineering teaching is promising, allowing an efficient form of evaluation.*

1. Introduction

The difficulty in establishing a clear division between good and bad software artifacts translates into a proportional difficulty in spotting peoples abilities and deficiencies towards the building of such artifacts. Organizations could benefit from this kind of profiling. By identifying the strengths and weaknesses of employees, managers could work on interventions to solve the weaknesses and to explore the strengths. Without a precise way of measurement, organizations need to guess the quality status of its internal staff, and this may lead to poor decision making.

This lack of criteria also affects the learning of software engineering concepts, which can vary according to the course instructor. This is particularly relevant in contexts where the learning should provide uniform results. For instance, organizations naturally desire that all employees participating in a training program end up with the same knowledge, regardless of when the training occurred and of who was responsible for the teaching.

Academic programs have a similar goal. Students who graduate should all share the same fundamentals of software engineering. In either case, it is hard to assure and control if the planned goals are reached, specially when the measurement is not objective.

In this work we propose a teaching methodology targeted at reducing this subjectivity. Our methodology is based on PDCA cycles, where the four stages (Plan, Do, Check, Act) are adapted to the context of teaching software engineering disciplines. We call this adaptation l-PDCA, where l stands for learning. As we shall explain throughout

the paper, one of the main premises of our proposal is the usage of checklists as objective instruments to evaluate students. With proper checklists defined, the evaluation can be done efficiently, reaching results that are well understood by both students and instructors. At the end of each l-PDCA cycle, the checklists can also provide useful information about existent profiles, such as human behavior and performance factors.

Additionally, the four stages surrounding the application of checklists enable a uniform learning experience, where the knowledge is equally distributed among the students. The l-PDCA approach can be considered a teaching framework. To implement it, one must consider the goals to be achieved, and prepare proper checklists to reach those goals. To demonstrate this idea, this paper presents a case study applied to an undergraduate course. The course is part of a software engineering program whose teaching methodology encourages the usage of Problem-Based Learning in any of its forms, such as the one described in this work.

This paper is organized as follows. Section 2 presents a brief introduction to PDCA and describes how PDCA was adapted to be used as a learning methodology. Section 3 presents the case study, reporting the results obtained. Section 4 discusses related work. Finally, Section 5 brings our concluding remarks.

2. The l-PDCA Teaching Methodology

The Plan-Do-Check-Act (PDCA) cycle is a management method used to control improvements or maintenances of processes. It originated from a Japanese interpretation to the ”Daming Wheel” - a modification of the Shewhart cycle [Darr 2007]. Figure 1 shows a PDCA cycle with its four basic stages. The purpose of the Plan Stage is to define the control goals and the means to achieve them, i.e., to establish the control directives (or procedures) to the management. The Do Stage aims at executing the tasks predicted in the Plan Stage and collecting data to process assessment. This stage requires training so the job is performed accordingly. The Check Stage proposes the evaluation of results through the comparison of the collected data with the established goals. The purpose of the Act Stage is to carry corrective actions to remove the anomalies found in the results, preventing them from happening again [Frakes and Fox 1996].

Figure 1. The PDCA cycles: Plan, Do, Check and Act stages and their purposes.

The PDCA cycle has been used to maintain or improve the process control directives. When used to maintenance, PDCA aims to comply with standard operating procedures, i.e., the process is repeated and the goal is a range of acceptable values. When used

for improvement, the process is not repeatable and the work carried out with a PDCA cycle aims at meeting a specific goal. In this case, the goal establishes a new control level for the organization [Jarvinen et al. 1998b]. The fact that a PDCA cycle is a management tool do not imply that it is used just by managers. All roles of an organization (directors, managers, technicians and operators) use the PDCA cycle as previously presented. However, technicians and operators are concerned about keeping the standard operating procedure; hence, they apply the PDCA cycle aiming maintenance.

On the other hand, directors and managers are concerned about establishing a new control level; hence, they apply the PDCA cycle aiming improvement [Jarvinen et al. 1998b]. We propose the use of PDCA stages as a way of teaching students about concepts related to software engineering and evaluating their learning. As mentioned, we named this adaptation l-PDCA, where l stands for learning.

While the focus of the original PDCA usually relates to verifying and improving the quality of a given process, the l-PDCA focus on establishing a teaching methodology that is able to spread knowledge and evaluate learning uniformly. Figure 2 illustrates the four stages of an l-PDCA cycle. The end of a cycle (an iteration) means that the teaching of some predefined knowledge units is done and a set of artifacts related to those units were delivered. The next sections explore the stages in details.

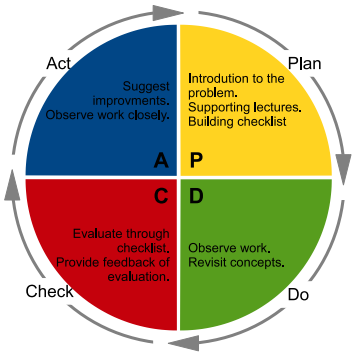

Figure 2. l-PDCA: adaptation of the PDCA stages for use on a learning environment.

2.1. Plan Stage

The purpose of the Plan Stage is to prepare the students for the work of the current iteration. The preparation introduces the students to the problem at hand, describes which artifacts should be delivered and explains the format of evaluation. Additionally, supporting lectures may be presented at this point, providing knowledge related to the task that should be accomplished. For instance, supposing that the project is currently at the *Analysis* iteration, the instructor declares that the goal of the iteration is to create a use case diagram. The students are also warned that the diagram would be evaluated according to its completeness and adherence to standards. To mitigate the risk of failure, the instructor talks to the students about good practices that apply in use cases creation.

At this stage, the instructor may also prepare an evaluation checklist (if none exists from a past instance of the teaching program). The items in the checklist should provide clear and unambiguous criteria to verify if the work is being performed as expected. Although the students know about the general topics of evaluation, they do not know exactly

what particular issues will be scored, so they should be prepared for everything that concerns the general topic. For instance, the *Analysis* iteration checklist could be composed by an item that checks if the use cases follow the naming convention detailed in the UML specification.

2.2. Do Stage

This is when the work actually gets done. The students know what the problem is and what needs to be done to solve it. At the end of the *Do* stage they are expected to deliver some software artifacts, as defined during the prior stage. At intermediary iterations, these may be sub-products of the project. At the final iteration, it will most likely be a complete product. The role of the instructor in this stage is reduced, so it becomes an observer most of the time. If necessary, the students can be aided with follow up lectures related to the concepts learned during the prior stage. However, we discourage guidance that is directly related to the specific problem the students are facing, since it may inhibit pro-active behavior and bias the evaluation. For instance, resuming the *Analysis* iteration example, the instructor should not mention that the name of a specific use case is incorrect, or that there are some important use cases missing. This sort of observations can be made at the *Act* stage, as we discuss later.

2.3. Check Stage

As the name suggests, this stage verifies the artifacts produced at the prior stage. After the evaluation is complete, the students become aware of the difference between the desired solution and what they actually accomplished. The instructor is responsible for providing feedback to the students about their work, so they can clearly understand the reasons of either failure or success. We propose the usage of checklists to evaluate software engineering learning.

Checklists are collections of items whose purpose is to verify if individual requirements of the work were done. The verification is binary. Each item can be marked as done or undone. The final score is measured by computing how many items of the checklist were done. It is desirable that the checklists allow instructor independent verifications, that is, the score obtained is the same regardless of the person who evaluated the artifacts. Large corporations can benefit from this sort of evaluation, since it helps homogenizing the knowledge among the employees, specially in aspects that involve internal standards and regulations.

Besides, it helps creating an atmosphere of fairness, since personal opinions would have no impact in the evaluation. This is particularly relevant in software engineering disciplines, in which there may be several solutions for the same problem, and the best solutions are usually underrated by the examiner. We emphasize that different checklists can be part of the same evaluation, measuring different skills. If a final grade is necessary, it can be a weighted composition of the individual checklists, with weights balanced according to the course goals.

The checklists can also be balanced according to a profiling strategy. In this case, the results are used to spot strengths and weaknesses. By identifying the weaknesses, it is possible to take corrective actions to overcome individual limitations. In large organizations, it is also important to identify people's strengths, so that their efforts can be channeled to functions where their skills are leveraged at their full extent.

2.4. Act Stage

With the knowledge of how much the delivered artifacts deviate from the expected solution, the students are able to perform the necessary changes to put the product back to its originally traced route. The *Act* stage is where these changes are made, allowing the next iteration to start with valid artifacts. We observe that the students may get stuck at some particular problems, having difficulties in deciding the proper strategy to solve it. In such cases, the instructor may become more directly involved in the project. The depth of the involvement depends on the instructor. It could range from a far support, such as using the Socratic Method to help students discover the root of the problem by themselves, to a close support, by proposing exactly what should be done to correct the defect. Naturally, this allows instructors to add their personal touch to the solution, which seems to violate the "lack of subjectivity" principle stated before. However, we believe this kind of direct intervention is needed in some cases, specially those where there is no standard rule that should be followed.

Besides, those situations generally imply the need of tacit knowledge, which is not easily found in manuals or books. There may be some relevant aspects not captured in the evaluation checklists, specially those that are not easily transformed into objective items. For instance, one item hard to verify is the existence of "code smells" in source code artifacts. It is important that all of those aspects are taken care of, even those that are not part of the evaluation, so they do not negatively impact the work at a later phase.

The *Act* stage ends a cycle, and if the students did their job, it assures that they all enter a new phase with valid artifacts. It does not mean the artifacts of different students will be the same. We emphasize that the existent artifacts are adapted rather than rebuilt from scratch. This opens the possibility of people working on different sets of artifacts, but that are in essence equivalent. At the end, these slightly different versions enable students to share ideas and to reason about the possibility of achieving the same result through different lines of thoughts.

3. Case Study

To evaluate our proposal, we report a case study developed within a course of the Software Engineering Undergraduate Program (SEUP) at the Federal University of Pampa (UNIPAMPA). The intention of SEUP is to prepare students for a variety of career options in both academic areas and software industry. The program is known for encouraging the usage of non-traditional teaching methods as an effort to enhance the quality of teaching. One of the approved initiatives, which actually defines the program – and differentiates it from most other programs – is the embracing of the Problem-Based Learning (PBL) methodology [Billa 2012].

In essence, PBL proposes a new teaching and learning process that provides means by which students can achieve a self-directed learning through the investigation of problems [Selçuk and Tarakçi 2007]. Throughout the academic life at the SEUP, students attend to several PBL based courses, one per semester. During a PBL-based course, they are introduced to a problem. The scope of the problem is defined according to the knowledge units that should be taught at the present semester [IEEE and ACM 2004].

The students are commonly divided in groups, and the course is given by one or more professors, which are called tutors. Each tutor is responsible for guiding a number

of groups. The number of groups per tutor is carefully decided, so tutors can dedicate enough time to their groups. Each group has the entire semester to elaborate and execute a project to solve the problem. To do that, students have to work with software engineering methods and techniques. Additionally, other tasks are required, such as teamwork, creativity thinking and pro-active research. On his turn, the tutor provides the necessary guidance and defines a calendar, considering the final presentation and checkpoints to evaluate the intermediary progress. Given this context, our goal was to introduce the l-PDCA as a way to implement the PBL methodology adopted at the SEUP.

3.1. Using the l-PDCA as the teaching framework for Problem Solving II

To implement the l-PDCA approach, we chose the `Problem Solving II` course (PSII), offered to undergraduate students in the second semester. The course project involved developing a central reservation system to be used for hotel management. The reasons for choosing this theme were manifold. First, students are relatively familiar with this kind of system (most of them probably already had the need to book a room in the past). The tutors were also familiar with the dynamics of a central reservation system, so they could easily come up with the software requirements and constraints.

41 students attended the course. They were divided in groups of 4 to 5 members, which resulted in 9 groups. Group members were chosen by applying a random selection. The outcome is the formation of arbitrary groups, which emulates a common scenario that happens in large corporations. Four tutors were designated to the course, resulting in the reasonable amount of 2 to 3 groups per tutor.

In the case study, tutors perform the instructor's role of the l-PDCA approach. The project was composed by three consecutive phases, namely Analysis, Design and Coding. In the first phase, the groups built a specification of the requirements. In the second phase, a class diagram was created, based on the requirements specification from the preceding phase. Finally, the class diagram was turned into functional code during the last phase. One of the most important parts of building the course was defining the checklists for each phase.

For this particular case study, the evaluation was more focused on the completeness of the solution found. In other words, we decided to evaluate how much of the expected product was delivered, instead of how well the delivered product solves the problem. The reason is that we were dealing with freshmen who just got started with the foundations of software engineering. It would be unfair to penalize them for ignoring good practices that only time and practice can teach. As we shall see, those quality related aspects were handled during the Act Stage, where the tutors feedback showed what could be improved and why specific changes were important.

To follow the l-PDCA principles, the checklists were composed beforehand by the tutors and kept hidden from the students. Since we focused on completeness, the students scored higher if they were able to cover most of the components the tutors expected them to cover. If needless/incorrect components were identified, the penalty was to ignore them completely.

We also observe that "Software Modeling and Design", "Database Modeling and Design" and "Human Computer Interaction" are courses only taught in the consecutive semester. Therefore, the project scope and complexity needed to be tailored according

to the students expected abilities. To do so, we striped the presentation layer and the persistence layer out of the project. Additionally, we only touched the surface of the full spectrum of UML software engineering models available, as we shall demonstrate. Next sections present details about the three phases of the project. For each phase, the four stages of l-PDCA are described, including the corresponding set of artifacts to be delivered and the checklists that verified their completeness.

3.1.1. Analysis Phase

At the Analysis Phase, the groups needed to establish a contract with the customer about the product that should be delivered, detailing precisely which requirements and constraints should be addressed. At the end, two documents were demanded: the use case diagram (UCD) and the requirements specification document (RSD). In what follows we describe how we implemented the four stages of the l-PDCA cycle in order to reach those goals.

Plan Stage: During this stage, the students received information related to the set of artifacts that should be delivered. Additionally, they got supporting lectures about Use Case Diagramming and Requirements Engineering, which included strategies to requirements elicitation and tools to document them. They also became aware that their work would be evaluated based on their ability to identify most of the use cases and requirements needed to implement the proposed system.

Do Stage: To enable a satisfactory requirements specification, the students were given a period of time to gain an understanding of the problem and the desired solution. It was up to the groups to acquire this knowledge, using the requirements elicitation techniques learned in class.

Check Stage: To measure completeness of the Analysis Phase, the UCD and the RSD were cross-validated against two different checklists: the Use Case Checklist and the Requirements Checklist. The Use Case Checklist verified if the necessary use cases were identified by the groups. According to the tutors' perspective, there was a total of 10 use cases. The Requirements Checklist verified if the necessary requirements were identified by the groups. According to the tutor's perspective, there were a total of 40 requirements, related to either functional requirements or business rules.

Act Stage: After the Check Stage, the groups knew how much of their job was done and how much was missing. During the Act Stage, they got a chance to add the missing use cases and requirements to the proper documents. Additionally, the tutors observed other problems with the delivered artifacts. The students were advised to improve their analysis to handle these issues. As expected, the most common problems identified relate to the correctness of the requirements. Some examples are provided below: (i) the meaning of the requirement was unclear and subject to interpretation; (ii) the requirement contradicted other requirement(s); (iii) implementation details were provided.

3.1.2. Design Phase

At the Design Phase, the groups needed to model the application for the project theme using software engineering fundamentals. At the end, a class diagram containing the underlying data model of the system should be delivered. In what follows we describe how we implemented the four stages of the l-PDCA cycle in order to reach this goal.

Plan Stage: During this stage, the students received information related to the artifacts that should be delivered. Additionally, they got supporting lectures about basic class diagram modeling and tools that allow UML diagramming. The students also became aware that their work would be evaluated based on their ability to identify most of the attributes and methods needed to implement the proposed system. They were also oriented to start their investigation based on the set of artifacts delivered in the Analysis Phase.

Do Stage: According to the tutors' orientations given in the prior stage, the groups built the class diagram following a bottom-up sequence of steps. At some specific moments, the tutors revisited concepts taught during the Plan Stage, providing further explanation about the bottom-up creation of the diagram. However, we observe that the concepts were illustrated without direct contact with the project theme. This is a borderline the tutors did not cross in order to enhance the learning experience.

Check Stage: To measure completeness of the Design Phase, the class diagram was cross-validated against two different checklists: the attributes checklist and the methods checklist. The Attributes Checklist verified if the necessary attributes were identified by the groups. According to the tutors perspective, there was a total of 60 attributes, spread across the classes of the model. For each attribute, we asked the groups to identify where in the class diagram it was located. The Methods Checklist verified if the necessary methods were identified by the groups. In essence, all requirements (functional requirements and business rules) needed to map to methods.

Act Stage: After the Check Stage, the groups knew how much of their job was done and how much was missing. During the Act Stage, they got a chance to add the missing attributes and methods into the class diagram. Additionally, the tutors observed other problems with the delivered diagram. As expected, the most common problems identified relate to the correctness/quality of the diagram. Some examples are provided below: (i) Methods with incorrect parameter lists; (ii) Incorrect associations between classes; (iii) High coupling and low cohesion.

3.1.3. Coding Phase

At the Coding Phase, the groups needed to transform the class diagram into source code. At the end, functional code related to the data layer of the system should be delivered. In what follows we describe how we implemented the four stages of the l-PDCA cycle in order to reach this goal.

Plan Stage: During this stage, the students became aware that their work would be evaluated based on their ability to build functional code. Additionally, supporting lectures were given about test coverage and how to identify test scenarios and test cases

from the UCD and the RSD.

Do Stage: This is when the source code actually got written. If we assume the groups had refined the class diagram considering the instructors suggestions, most of the work of this stage involved filling the gaps of the skeleton code. The students were encouraged to write good code from the start, testing the functions and business rules as they got implemented. To promote this idea, the tutors introduced notions about Test-Driven Development.

Check Stage: To measure completeness of the Design Phase, the source code was cross-validated against a Test Case Checklist. The Test Case Checklist verified if the code passed a number of different cases, involving some input values and an expected output. Of course, mapping all possible test cases is an unrealistic goal. For the sake of the evaluation, we restricted the set of test cases using a simple rule. For each use case, there should be one test case where the use case executed successfully and one test case for each constraint where the constraint was not satisfied.

Act Stage: After the Check Stage, the groups knew how much of the proposed test cases failed in the execution. In most cases, the failure occurred because the constraints were not properly handled in the code or even because the use case was not implemented at all. Either way, the students were warned about what should be done in order to correct the flaws. We observe that there was no need to adjust the code, since this was the final phase of the project. Instead, the tutors used the time available during the stage to point out other issues with the code, such as those related to static analysis and usage of patterns. Additionally, the students became aware that the test cases evaluated by the tutors were far from a comprehensive list of possible situations, and much more testing was necessary in order to deliver a high quality product.

3.2. Results

This section reports the observations made when compiling the information gathered from the course. Figure 3 sums up the results achieved, by presenting the general performance of the students at each checklist.

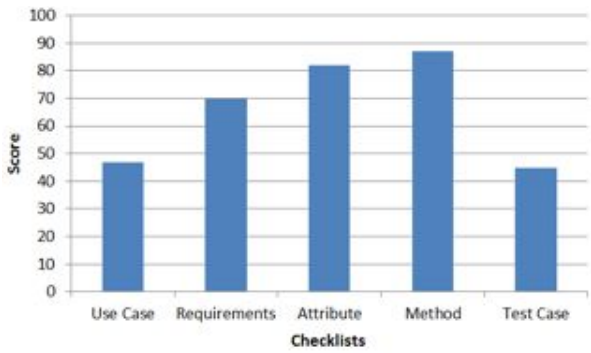

Figure 3. Results Related to the Students Performance on the Checklists.

The first conclusion is that the groups did a better job at the first two phases (Analysis and Design). This shows that programming skills needed improvement, which was expected for first year students. Observing the checklists of the Analysis Phase (Use Case

and Requirements), we also concluded that students faced difficulties in capturing the requirements, which indicates a need to enhance communication skills. Additionally, even though the students score were basically a matter of making sure the job was done, and not how well it was done, still their overall performance was relatively low, as depicted in Figure 3. This indicates that, in general, the students lack maturity to conclude work where the goals are well-defined and easy to understand. These results were reported to the SEUP coordination, so that corrective measures could be taken, such as applying motivational lectures, explaining that dedication and perseverance are important factors in the educational growth.

It is important to remark that a higher number of checklists could be used to identify a more diverse range of profiles. As stated before, the number and type of checklists should be decided based on what qualities need to be mapped. In our case, considering the course is given for students in the first year of the program, the intention was to identify the level of effort of the students rather than their intellectual qualities.

We could not analyze if the proposed checklists are suited as instructor independent ways of evaluation. Since there was a disjoint relation between tutors and their groups, it was not possible to compare the checklists results among themselves. As future work, we intend to apply the same checklists, but having different reviewers for the same group. Nevertheless, we observe that the l-PDCA methodology managed to be an efficient way of evaluation which the tutors felt comfortable with. The items of the checklists were discussed in loco with the students, so that questions regarding the work could be solved quickly. Besides, according to the tutors, the students considered the evaluation approach to be fair and acknowledge the problems found in their software artifacts.

4. Related Work

Many researches adopt PDCA cycles to control quality in organizations. It is a logical working process that can be adapted to improve quality in different contexts. For instance, Ning et al. [Ning et al. 2010] adopted the PDCA cycles to provide continuous improvements in software quality. The model helped to take objective decisions, to identify demands of software adaptation, and to organize task adjustments.

Furthermore, PDCA can also be used in education context. The model can be directly used to validate teaching-learning methods as proposed by Fuhou [Fuhou 2009]. According to this work, professors plan based on their experience and fall back in teaching theories aiming to solve learning problems; they regard curriculum goals, students knowledge and university support conditions to define the Do and Check stages; and, finally, they learn lessons and develop standards to continuously improve teaching quality in the Act Stage.

On the other hand, PDCA cycles can be adapted according to the research goals. For instance, Cukusic [Cukušić et al. 2010] proposed the design of an e-learning process management (planning, organizing/implementing, controlling and improving) from the PDCA generic model. Our work proposes the usage of the PDCA cycles associated with checklists to make clear the goals of each stage and to provide a uniform evaluation/learning methodology. Jarvinen et al. [Jarvinen et al. 1998a] proposed a similar integration of checklists and PDCA cycles to manage a developing team, in which checklists act as working assistants to ensure the corporation goals are captured into the PDCA

stages.

5. Conclusions

This paper presented an approach to software engineering teaching that is completely based on PDCA cycles. The approach, named l-PDCA, adapts the Plan, Do, Check, and Act stages to conform with teaching scenarios, rather than the management scenarios of the regular PDCA. Besides providing students with a uniform learning environment, this adaptation targets at uncovering skills that are not easily identified during training courses.

To demonstrate the approach, we conducted a case study with a class from the Software Engineering Undergraduate Program (SEUP) at Unipampa. One of the results we achieved was showing that l-PDCA fits the PBL methodology applied at the SEUP. The role of the students and the tutors match the role of students and instructors of l-PDCA. The first is responsible for solving a problem. The second is responsible for providing the tools and the knowledge necessary to solve the problem. Besides, the l-PDCA addresses a shortcoming of having many tutors for the same instance of the course, which is the heterogeneity in the learning and evaluation process.

As stated earlier in the paper, checklists are the core of l-PDCA. Ill-defined checklists may lead to failure, frustrating students for being unable to associate the evaluation with the knowledge units taught, and also instructors for being unable to identify students' strengths and weaknesses related to those knowledge units. In this context, our case study showed that the methodology was approved by students and tutors. Students were able to easily understand what went right or wrong with their deliveries, acknowledging the outcome. On their turn, the tutors were able to evaluate the students quickly. Besides, the composition of the checklists enabled some basic profiling, by spotting which skills had an urgent need of improvement.

What we tried was to break the deliverables into atomic units of work, so the completion of each unit could be objectively evaluated. We recall that the PBL approach values teamwork, so students need to work together during the course. In this context, the checklists were used to perform a collective evaluation. In the future, we also intend to evaluate the components of a group, so that individual profiles can be uncovered. This can be achieved by breaking down a product into modules, and delegating one module per student.

Even though we demonstrate the application of the l-PDCA for an academic course, the same ideas can be explored in any kind of training program, such as those sponsored by companies that want to prepare their employees for specialized assignments. Additionally, the form of evaluation proposed applies smoothly to products/artifacts related to software engineering, but are not restricted to this area. With the proper adjustments, the general l-PDCA framework can contribute in the learning/profiling of other areas as well, specially when subjectivity is an issue.

References

[Billa 2012] Billa, C. Z. (2012). Experiência de APB aplicado em engenharia de software. In *Anais do Internacional Conference PBL-ABP 2012*, Santiago de Cali, Colômbia.

[Cukušić et al. 2010] Cukušić, M., Alfirević, N., Granić, A., and Garača, e. (2010). e-learning process management and the e-learning performance: Results of a european empirical study. *Computer & Education*, 55(2):554–565.

[Darr 2007] Darr, K. (2007). Quality improvement: The pioneers. *Hospital Topics*, 85(4):35–38. PMID: 17405423.

[Frakes and Fox 1996] Frakes, W. and Fox, C. (1996). Quality improvement using a software reuse failure modes model. *Software Engineering, IEEE Transactions on*, 22(4):274 –279.

[Fuhou 2009] Fuhou, Z. (2009). Pdca circulation in university education applied research. In *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*, ICISE '09, pages 3375–3378, Washington, DC, USA. IEEE Computer Society.

[IEEE and ACM 2004] IEEE and ACM (2004). *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. IEEE and ACM.

[Jarvinen et al. 1998a] Jarvinen, J., Perklen, E., Kaila-Stenberg, S., Hyvarinen, E., Hyytiainen, S., and Tornqvist, J. (1998a). Pdca-cycle in implementing design for environment in an r & d unit of nokia telecommunications. In *Proceedings of the 1998 IEEE International Symposium on Electronics and the Environment (ISEE-1998)*, pages 237–242.

[Jarvinen et al. 1998b] Jarvinen, J., Perklen, E., Kaila-Stenberg, S., Hyvarinen, E., Hyytiainen, S., and Tornqvist, J. (1998b). Pdca-cycle in implementing design for environment in an r amp;d unit of nokia telecommunications. In *Electronics and the Environment, 1998. ISEE-1998. Proceedings of the 1998 IEEE International Symposium on*, pages 237 –242.

[Ning et al. 2010] Ning, J., Chen, Z., and Liu, G. (2010). Pdca process application in the continuous improvement of software quality. In *2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering (CMCE)*, volume 1, pages 61 –65.

[Selçuk and Tarakçi 2007] Selçuk, G. and Tarakçi, M. (2007). Physics teaching in problem-based learning. In *AIP Conference Proceedings*, volume 899, page 844.