RoboCode & Problem-Based Learning: A non-prescriptive approach to teaching programming

Jackie O'Kelly
NUI Maynooth, Ireland
jackie.okelly@gmail.com

J. Paul Gibson
NUI Maynooth, Ireland
pgibson@cs.nuim.ie

ABSTRACT

The fundamental principle behind Problem-based Learning (PBL) is that the problem is the driving force that initiates the learning. In order to function effectively in a PBL environment a *good* set of problems is required. Solving problems is a vital element within Computer Science and yet the discipline has been slow to embrace PBL as an approach to learning. The net result means that there are few *good* PBL problems available to assist new practitioners with implementation. PBL emphasizes a real-world approach to learning, and we present a RoboCode Competition as a candidate for a *good*, realistic PBL problem within the computer science discipline. We list and identify the criteria that categorise a PBL problem as *good* and validate the RoboCode domain against these criteria. We argue that the concept of *freedom* — in different guises — plays a key role in making PBL a *good* mechanism for teaching programming, and for making RoboCode a *good* domain for PBL.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; I.2.6 [**Learning**]: Misc.; D.2 [**Software Engineering**]: Misc.

General Terms

Problem Based Learning

Keywords

Computing Education Research

1. INTRODUCTING PBL AND CS1

Programming is a fundamental skill that all Computer Science (CS) students are required to learn. However, programming courses are generally regarded as difficult, and often have the highest dropout rates[17]. A psychological overview of programming pedagogy by Winslow[20] reports that: novice programmers neglect strategies, are limited to surface knowledge of the subject, and that this knowledge is fragile. Fragile knowledge is described as something the student knows but fails to use when necessary. Jenkins[11] argues that a surface approach to learning is inadequate in such a practical subject and advocates a deep,"learn by doing" approach.

Our experience in first year programming (CS1) in the CS department at NUI Maynooth (NUIM) reflects the issues with shallow learning. In order to address this, we implemented a Hybrid Problem-based Learning approach to the CS1 module in the academic year 2003-04: the implementation of this work is reported by O'Kelly[12], with analysis reported by O'Kelly and Gibson[13].

Within the PBL environment the problem acts as the catalyst that initiates the learning process. It is said that this way of learning encourages a deeper understanding of the material, rather than surface learning, because it is the students who are actively "doing". As the problem is such a critical component of the learning process it is imperative that one uses *good* problems. In 2001, Duch identified five characteristics of what makes a PBL problem *good*[7]:

1. Effective problems should engage the students' interest and motivate them to probe for deeper understanding of the concepts being introduced.
2. PBL problems should be designed with multiple stages.
3. The problems should be complex enough that cooperation from all members of the group will be necessary in order for them to effectively work towards a solution.
4. The problem should be open-ended.
5. The content objectives of the course should be incorporated into the problems.

One of the major stumbling blocks to the implementation of PBL within any discipline is the lack of a good set of problems. However, with the exception of a few disciplines, good PBL problems usually do not appear in textbooks[18]. Clearing houses provide an avenue to allow for the sharing of problems, but unfortunately there is a lack of CS problems[1]. This could be partially explained by the small number of reported PBL practitioners in the CS discipline[14].

[1]One could argue that there is an abundance of CS programming problems available; however, the vast majority of these problems place emphasis on the learning of a particular programming concept rather than problem solving.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE'06, Univ. of Bologna, Italy, 26 - 28 June 2006.
Copyright 2006 ACM ? ...$5.00.

2. THE ROLE OF FREEDOM

In this section we discuss the role of freedom in education. We take cognizance of the fact that while a price must be paid for this freedom, our role as educators within the CS discipline, is to create an environment (particularly for first year students) that allows them the freedom to embrace the explorative, creative process of solving problems.

2.1 Freedom in education

It is our view that in tertiary education students are encouraged to think for themselves, express their views, question, discuss and be free to disagree in their pursuit of knowledge. However, our methods of instruction span a continuum from the prescriptive to the chaotic, with both extremes working against the student. When we are prescriptive we leave no room for the student to seek knowledge, we see him as the empty vessel — or the blank slate — and our job is to fill the vessel or to write on the slate. At the other extreme we provide little or no structure to guide the student in developing their knowledge. A number of positions exist along this continuum where we can provide scaffolding for the students and still allow them the freedom to seek knowledge. The old adage that one learns from one's mistakes allows for creativity, experimentation, and reinforces the belief that making a mistake is not *always* wrong.

It is our experience that it is very difficult to find a problem that is *just stable enough* in the sense that — for a wide range of students — the balance between being non-prescriptive and open, but not so open as to be chaotic, is assured. We argue that the RoboCode problem fulfils this stability criteria.

2.2 Freedom in CS and SE

Software engineering is not constrained by the physical laws and engineering principles that govern other engineering disciples. Within this unconstrained environment, software engineers manage the complexity of choice through consistent application of fundamental conceptual tools such as abstraction; this allows them to continually improve the software engineering process - "re-writing the rules" for development, making the process as much an art as a science." However, a price must be paid for this freedom, which is why design and quality management are such important issues for software engineering. The true role of design is to create a workable solution to an ill-defined problem, thus software design is a creative process that cannot be reduced to a routine procedure, it involves discovery and invention, and it frequently requires intuitive leaps between abstraction levels[10].

2.3 Freedom in CS1

The majority of students entering CS1 (in Ireland) believe that because "learning by rote" was successful for them in the past, it will continue to be successful. However, learning to program is not suited to rote learning: it requires the student to understand the problem, develop a solution, implement this solution in a programming language, compile the program, develop test data, test the program, and iterate the debug, compile, test cycle until they have a working solution. Researchers have found that a student's learning style can affect their performance in introductory computer science courses[5]. In order to facilitate these diverse learning styles we need to create an environment that allows students the freedom to divest themselves of the necessity they feel to rote learn and to embrace the explorative, creative process of solving problems. They must learn that experimentation is fundamental to all scientific disciplines, and they must become proficient in their use of a computer as their laboratory for designing and implementing computer science experiments.

3. ROBOCODE

3.1 Background and Motivation

The retention of first year students in CS is a concern for universities globally. In order to improve retention in our University we tried to create an environment that was both fun and educational. When we were approached to help develop a National RoboCode Competition[2], both authors saw potential benefits to enhance their PBL initiative and add an interesting challenge for their students. The competition was aimed at first year programming students, allowing them to demonstrate their programming abilities, advance their knowledge and have some fun. The fun transpired to be what Papert[15] termed "hard fun"; in that it is both challenging and interesting, and this implies "hard". Barrett[1] argues that learning in a PBL environment demands both the fun of playing with ideas and the hardness of refining and reworking ideas, and that both complementary parts are needed for learning. In order for any of this to occur we need the right type of problem to support the process.

3.2 Technical Information

RoboCode[3], developed and released by IBM in 2001 is a programming game (and development environment) that teaches Java in a fun, rewarding manner. The game environment consists of a small rectangular battlefield populated with robot-tanks. These cleverly programmed graphical robot-tanks manoeuver, search and destroy other robot-tanks on screen. The skeleton of a robot-tank is provided in the class Robot; in addition, sample robot-tanks that perform a specific type of action are supplied. Students can look at how a robot-tank performs and peruse the corresponding Java code. They can combine actions from a number of robot-tanks and immediately test out the performance of the new robot-tank in the battle arena. Bonakdarian & White[3] argue that RoboCode promotes "Learning by subversive means" where the students have to learn in order to "play" and are motivated by wanting to participate. Loadholtes[4] and Hartness[9] promote RoboCode as a tool for teaching Artificial Intelligence (AI); while learning about AI is not a learning outcome of the CS1 curriculum, students must consider tactics for survival so that their robot-tanks behave and respond "intelligently" to their environment.

4. ROBOCODE IN THE CLASSROOM

The RoboCode Competition is introduced to students midway through the year. Students will have covered half the curriculum, thereby giving them knowledge of fundamental

[2]Phil Bourke, Tipperary Institute of Technology, Ireland initiated the National RoboCode Competition.
[3]RoboCode Central - `http://RoboCode.sourceforge.net/`.
[4]Loadholtes, N., (2004) *IBMs RoboCode: A Platform for Learning AI*, `http://ai-depot.com/Essay/RoboCode.html`.

programming concepts: variables, assignment, control flow, iteration, functions (methods), etc... Students' abilities to transfer what they have learned to new situations provide an important index of adaptive, flexible learning[4]. RoboCode is implemented using event-based programming and more advanced programming features that the students have not (yet) been exposed to.

4.1 Engaging the students' interest

It is well-accepted that effective problems engage the students' interest and motivate them to probe for deeper understanding of the concepts being introduced. It is also known that the students' initial reactions to a subject or topic is critical to them gaining an interest. Consequently, it is important that the students are introduced to the RoboCode problem in an appropriate manner. We chose to do this by giving them an initial statement that emphasises the competitive nature of the problem:

> *Develop a robot-tank capable of beating all its competitors and winning the competition. See* `www.cs.nuim.ie/RoboCode` *for guidelines and rules of the competition.*

Engaging the students' interest or sowing the seeds of interest requires that students become stakeholders in the problem scenario.

4.2 The need for team-work

The decision to enter the competition is an individual student's; however, the rules of the competition dictate that a team can include up to three people. In many programming situations, the primary working unit is a team, not an individual[19]. Constructing a team is part of the problem scenario; students have prior knowledge of how teams work as they have been participating in teams during semester 1.

DeMarco & Lister[6] describe the concept of a "jelled team" as a group of people so strongly knit that the whole is greater than the sum of the parts. They argue that once a team begins to jell, the probability of success goes up dramatically — they don't need to be motivated as they already have (learning) momentum. We encourage students to find team members with complementary skills and roles as, for example: a strategist, a programmer, a mathematician, a designer, a documenter, a tester etc... If a team combines the right balance of skills then their learning and opportunities for success increase, as each member has a specific talent to contribute.

We require that all teams submit an entry form in order to partake in the internal competition; the winners of the internal competition go forward to represent the University at the national competition. From the moment a team of students submit their entry form to the internal competition they have become stakeholders.

4.3 Importance of experimentation

All teams start by playing around with the sample robot-tanks in the standard RoboCode libraries. The students observe how the robot-tanks perform in a game, look at the code and try to understand how it works. Then, the students make small changes to one of the robot-tanks and observe and document the results, asking: does the robot-tank perform better or worse due to the change? One of the features of RoboCode is that the results of one's efforts can be seen instantly, this "instant gratification" speeds up the development learning cycle and is analogous to the "prototyping model" used in software development.

Experimentation — for gaining understanding — is key to the software prototyping process. It is often used when: a customer has a general set of objectives, but is unclear on the detail; or if a developer is unsure of the efficiency of an algorithm, or the adaptability of an operating system; or there is debate about the form that the human-machine interaction should take; etc... The process begins with requirements gathering where the developer and customer meet and define the overall objectives for the software. A "quick design" then leads to the implementation of a prototype. This prototype is evaluated by the customer (user) and is used to refine the requirements for the software to be developed[16]. This process structure is analgous to the way in which RoboCode teams work in order to develop their final "product".

Students become enraptured by their ability to make the robot-tank "fun", "weird", "strange", etc...; initially, they alter the appearance of the robot-tank's colours and make the colours synonymous with their team name. This naturally progresses to them wishing to alter their tank's behaviour. Students are willing to experiment: fear of failure is not an issue; if what they try does not work they go back to an earlier version and try again (if their implementation was incorrect with respect to their design) or try something different (if their design did not prove to be an improvement on previous models).

4.4 Re-Use

Once the students have gained an understanding of how the robot-tanks work they set about researching how other programmers have developed robot-tanks. There is a plethora of robot-tanks and code available on the Internet; there are numerous sites that explain how to develop robot-tanks through the use of on-line tutorials, or tips-and-tricks. At this stage in their academic careers, this provides an ideal opportunity to introduce the notion of plagiarism and its relevance to software re-use[5].

The students are required to acknowledge any piece of code (or design) that they use but that they did not develop themselves. A check is run on the code submitted and the penalty for not adhering to the rule is disqualification. Bierre & Phelps[2] cite this proliferation of code as the reason they implemented a variant of RoboCode that addresses the availability issues. To date, students have understood the seriousness of plagiarism, and this is a valuable lesson to learn early in one's study or career.

Once the students have completed their research, which is in effect their requirements gathering, they set about designing their robot-tank. The design incorporates the strategy that they will use for the different stages of the competition. Implementation is then undertaken; this requires the students to write Java code for their robot-tank. Testing requires the students to ensure that their robot-tank is robust, can compete against and defeat all robot-tanks given as examples (usually by the lecturer). Finally, students are

[5]A technical report is used to guide students in their final year projects (*Software Reuse In Final Year Projects: A Code of Practice*, `NUIM-CS-TR2003-12`) and the guidelines in this report are referred to — where appropriate — in all their programming and software engineering modules.

also encouraged to test their tanks in competition against a random selection of robot-tanks (both strong and weak) taken from the Internet. These steps are applied iteratively, resulting in a product that should meet its requirements. This model of development called the "Incremental model" combines elements of the linear sequential model (applied repeatedly) with the iterative philosophy of prototyping[16].

5. ROBOCODE — THE COMPETITION

In the previous section we have shown how RoboCode — through its practical application in the classroom — meets two of Duch's 5 characteristics for a good problem. In this section we look at how the remaining 3 criteria — multiple stages, open-endedness and the incorporation of course objectives — are met by the RoboCode competition. We then argue that the RoboCode competition also addresses the additional criteria of needing to find a balance between order and chaos in the learning process.

5.1 PBL should have multiple stages.

The RoboCode Competition naturally provides a problem with multiple stages. The internal competition is run under the same rules as the national competition. Each team must submit their robot-tank, code and documentation prior to the competition date. A panel of experts (usually a selection of academic staff from the participating institutions) interviews teams about their submission. Any member of the team can be called upon to answer a question. This allows students the opportunity to explain their strategy and defend their work, improving their oral presentation skills.

The competition consists of a league, a semi-final and a final. Within the league section, each team's robot-tank is placed in a group. Points are awarded to robot-tanks based on their performance during each one-on-one match: 2 points for a win, 1 point for a draw. Within each group, every team plays against every other team.

The robot-tanks that accumulate the most points in each group progress to the next level in the league. Here, teams are allowed to refine their robot-tank (alter their code); they can change the behaviour of their robot-tank between levels in order to counter the other robot-tanks they are competing against. All teams will have seen every robot-tank in action once the first level is complete; the decision to gamble with the existing robot-tank or alter its behaviour can be compared to decisions required by businesses competing in the same market segment. The main decision is whether to stay with what they know is "working" and hope that the status quo is maintained or to implement a change and gamble for better (or worse)?

5.2 Complexity and open-endedness

This decision-making also highlights Duch's argument[7] that problems should be complex enough that cooperation from all members of the group will be necessary. A balanced, well-jelled, team will have the skills to make the best call in this situation. The problem is certainly open-ended, as no two robot-tanks have the same behaviour and it is difficult to predict how each opposing team will change, if at all, the behaviour of its robot-tank in the following round.

The league continues in a knockout fashion until only four teams remain and these go forward to the semi-final. Once again, teams are allowed to refine their robot-tanks prior to the semi-final. The semi-final consists of 3 one-on-one matches. Each match runs back-to-back so no alterations to the robot-tanks are allowed once the semi-final begins.

The finalists are allowed to refine their robot-tanks prior to the final. However, in the final they not only compete against the other finalist but also against five "pit robot-tanks"[6]. The behaviour of the robot-tank that allowed it to qualify for the final may not be the best behaviour in a mêlée type situation. Once again the students face a complex and open-ended problem. The objective for each robot-tank finalist is to survive longer in the mêlée than its opposing robot-tank finalist.

5.3 Course content objectives

This problem not only explicitly incorporates content objectives of the course but also challenges students to reach beyond them. The problem activates prior knowledge — both the knowledge from previous experiences, which everyone brings to their learning, and the knowledge of programming which the students have covered in semester 1, which for some students is "fragile". Through the course of *RobotCoding*, this fragile knowledge becomes concrete so that the learning becomes deep(er).

The students use and apply software engineering cognitive tools: refinement, sub-classing, re-use and genericity[8]. The students employ the software development process of requirements analysis, design, implementation, testing, re-design, re-code, and re-test; and this is a critical part of a software engineer's education. The ability to document their work — both in a technical and non-technical manner — is a requirement not only for their academic life but also for their work life; as is the skill to present, explain and defend their work in a coherent manner. Strong learners can explain which strategies they used to solve a problem and why they used them. The skill to work effectively in a team environment is a requirement in today's global economy. One of the major advantages of this problem is the cohesiveness it brings to the team, which increases the quality of its work.

5.4 Freedom: balancing order and chaos

Software engineering is all about going from *what* to *how*, moving from abstract problems to concrete solutions. This involves design steps: decisions that are made in order to move a model away from an open (non-deterministic and non-executable) description of requirements to a closed (deterministic and executable) description of the implementation. Software engineering (design) is difficult because it cannot, in practice, be done prescriptively (otherwise we would automatically generate solutions from problems); and it cannot be done in a purely ad-hoc fashion (otherwise we would not need software processes to manage the complexity of the systems and behaviour being modelled). Software engineering is a unique mix of science, engineering and art: the best practitioners know that each new project (problem) requires a different balance between the potential chaos of innovation and the constraints imposed by order and structure.

The freedom to find the correct balance was an important part of the RoboCode learning experience. In the 1st year of our participation in the national competition, our

[6]These are usually contributed by academic staff and are intended to provide very stiff competition for the students' tanks so that their most realistic objective is to maximise survival time rather than to actually defeat the pit robots!

winning team chose a very conservative approach to constructing their robot: they combined — in a reusable, prescriptive manner — the behaviour of robots that they had found on the Internet. They came 3rd in the national competition. In the 2nd year, our winning team took a very innovative, ambitious approach with little reuse of existing models. They worked in an experimental, ad-hoc fashion which suited the profile of the team members. They came 2nd in the national competition. We believe that each of the teams benefited from having the freedom to choose how they would work, within the constraints of the competition rules. (We note also that the two finalists were teams made up of 3 members — the maximum allowed.)

RoboCode also provides freedom to the lecturers to experiment with the balance between order and chaos. It is very easy to introduce new competition rules that constrain the way in which the students can work, making the development process much more prescriptive. It is also straightforward to weaken the rules so that students have much more freedom to experiment.

6. CONCLUSIONS

Competition is an everyday occurrence in the real world and effective problems in PBL emphasize this real world aspect. The RoboCode problem presented in this paper combines elements of fun, programming, games, AI and competition. It encourages the fun element of creative ideas within the constraints of the RoboCode environment with the challenges of refining these ideas into a workable solution. We would argue that this problem transforms fragile knowledge into a concrete transferable skill that can be applied in new situations. Students develop skills for each stage of the software development process: requirements analysis, design, implementation, and testing; and they can think critically, reflect on their work, conduct tradeoffs and make informed decisions. Our experience shows us that in order for students to gain the maximum benefit from this problem they should have prior experience of working in a team environment. Within PBL the focus is shifted from teaching to learning and this shift in conjunction with a good problem (RoboCode) provides each student with the freedom to think for themselves, activate their prior knowledge and acquire new knowledge in an explorative and creative way.

7. REFERENCES

[1] T. Barrett. Who said learning couldn‘t be enjoyable, playful and fun? – the voice of PBL students. In *PBL in Context — Bridging Work and Education*, pages 159–175. Tampere University Press, 2005.

[2] J. Bierre and A. Phelps. The use of MUPPETS in an introductory Java programming course. In *SIGITE04*. ACM, October 2004.

[3] E. Bonakdarian and L. White. Robocode throughout the curriculum. *J. Comput. Small Coll.*, 19(3):311–313, 2004.

[4] J. Bransford, A. Brown, and R. Cocking, editors. *How People Learn: Brain, Mind, Experience and School*. National Research Council, 2000. Committee on the Development in the Science of Learning and Committee on Learning Research and Educational Practice, Commission on Behavioural and Social Science Education, National Research Council, Expanded ed.

[5] A. Chamillard and R. E. Swardl. Learning styles across the curriculum. In *ITiCSE 2005: Proceedings of the 10th Annual SIGCSE Conference Innovation and Technology in Computer Science Education*, pages 241–245, New York, NY, USA, 2005. ACM Press.

[6] T. DeMarco and T. Lister. *Peopleware (2nd ed.): productive projects and teams*. Dorset House Publishing Co., Inc., New York, NY, USA, 1999.

[7] B. Duch. *Writing Problems for Deeper Understanding*, pages 47–53. Stylus Publishing, Sterling, Virginia, 2001.

[8] J. P. Gibson and J. O'Kelly. Software engineering as a model of understanding for learning and problem solving. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 87–97, New York, NY, USA, 2005. ACM Press.

[9] K. Hartness. Robocode: using games to teach artificial intelligence. *J. Comput. Small Coll.*, 19(4):287–291, 2004.

[10] W. S. Humphrey. *PSP: A Self-Improvement Process for Software Engineers*. Pearson Education, Inc., NJ, 2005.

[11] T. Jenkins. A participative approach to teaching programming. *SIGCSE Bull.*, 30(3):125–129, 1998.

[12] J. OKelly. *Designing a Hybrid PBL Course: A Case Study of First Year Computer Science in NUI, Maynooth*, pages 43–53. CELT, NUI Galway, Ireland, 2005.

[13] J. OKelly and J. P. Gibson. PBL: Year one analysis — interpretation and validation. In *PBL In Context Bridging work and Education*, 2005.

[14] J. OKelly, R. Monahan, J. P. Gibson, and S. Brown. Enhancing skills transfer through problem-based learning. Report nuim-cs-tr-2005-13, Department of Computer Science, National University of Ireland, Maynooth., 2005.

[15] S. Papert. *The Connected Family : Bridging the Digital Generation Gap*. Longstreet Press, October 1996.

[16] R. S. Pressman. *Software Engineering: A Practitioner's Approach w/ E-Source on CD-ROM*. McGraw-Hill Science/Engineering/Math, November 2001.

[17] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[18] C. Tien, S. Chu, and Y. Lin. Four phases to construct problem-based learning instruction materials. In *PBL In Context Bridging work and Education*, pages 117–133. Tampere University Press, 2005.

[19] G. Weinberg, editor. *The Psychology of Computer Programming*. Dorset House Publishers, New York, 1998.

[20] L. E. Winslow. Programming pedagogy – a psychological overview. *SIGCSE Bull.*, 28(3):17–22, September 1996.