American Journal of Educational Research, 2013, Vol. 1, No. 8, 307-312
Available online at http://pubs.sciepub.com/education/1/8/7
© Science and Education Publishing
DOI:10.12691/education-1-8-7

Competing Dichotomies in Teaching Computer Programming to Beginner-Students

David Nandigam[1], Hanoku Bathula[2,*]

[1]Department of Technology, Northcote College, Auckland, New Zealand
[2]Graduate School of Management, The University of Auckland, Auckland, New Zealand
*Corresponding author: hanoku@outlook.com

Received January 03, 2013; Revised September 12, 2013; Accepted September 15, 2013

Abstract The goal in teaching computer programming is to develop in students the capabilities required of a professional software developer. Beginner programmers suffer from a wide range of difficulties and deficits. Several studies suggest that undertaking computer programming for meeting a real industry application is still a challenge for many students even after studying for a year or two. The purpose of this paper is to investigate the challenges in teaching computer programming to beginner-students and to initiate a dialog in the information and communication technology teaching community on how to teach and assess computer programming courses effectively. We undertake an extensive literature review to identify four major programming dichotomies in teaching computer programming: knowledge versus application, comprehension versus generation, procedural versus object oriented and functional versus imperative. Further, based on our teaching experience, we propose a practical approach to teaching computer programming to beginner-students. The paper discusses the implications to ICT teaching community and how teaching and assessments can be made effective to achieve the goal of making beginner programmer learn not only knowledge but also relevant application skills. We believe that the study would contribute to making ICT teaching more practical and effective in achieving their educational goals.

Keywords: *teaching strategies, computer programming, beginner students, information technology curriculum, dichotomies*

Cite This Article: David Nandigam, and Hanoku Bathula, "Competing Dichotomies in Teaching Computer Programming to Beginner-Students." *American Journal of Educational Research* 1, no. 8 (2013): 307-312. doi: 10.12691/education-1-8-7.

1. Introduction

"*It's not what you know that counts anymore. It's what you can learn and do*" Don Tapscott.

Computer programming with the specific aim of meeting an authentic need is the skill that computer science students are expected to master. However, learning to program is quite challenging for beginner programmers as they suffer from a wide range of difficulties and deficits [1,2] conceptual 'bugs' [3,4] and misconceptions [5]. Considering the problems programming students are expected to solve in a programming course, de Raadt et al [6] developed a three-level scale: 'system', 'algorithmic' and 'sub-algorithmic'. Problems at sub-algorithmic level may look simple because they do not involve algorithms or system designs. Examples of problems of this scale include avoiding division-by-zero, achieving repetition until a sentinel is found, and so on. Strategies used to solve problems at this level are particularly relevant to beginners in their initial exposure to the programming process. Yet, these strategies are also a fundamental part of solving problems at any level.

Unfortunately, several studies suggest that undertaking computer programming to meet a specific real application is still a challenge for many students even after studying it for one or two years. Lister et al [7] attributed poor results to poor problem-solving ability in students. The BRACElet project conducted at Auckland University of Technology showed that many students exhibit a fragile programming knowledge and very few can demonstrate clear understanding of programming strategy [8].

We are experienced academics teaching students from various countries and backgrounds. In spite of our experience, we are amazed at the challenges that our students face in their learning. So, we want to examine in this paper why teaching computer programming is still a challenge, even after over 40 years since it was first identified [9]. We believe that this would initiate a dialog in the ICT teaching community on the effectiveness of teaching and assessment approaches in beginner courses in computer programming. For this purpose, we undertake an in-depth literature review and combine it with our personal knowledge to identify the practical problems of teaching computer programming to beginners. Our study is divided into four sections. The next section examines the blurriness of goals of teaching computer programming, and then identifies four competing dialectic programming dichotomies, followed by an integrated practical approach to teaching computer programming. The last two sections are discussions followed by conclusion.

2. Blurry Goals of Teaching Computer Programming

Unlike a decade ago, the field of information and computer technology (ICT) is invaded by other conventional disciplines. Also the question is whose domain is ICT – engineers, scientists, technologists, commerce graduates or linguists? Practically, anyone could wander in. It would be revealing to look at the context of stage 1 of Bachelor's IT Programme, where we find that participants come from a variety of backgrounds. They differ linguistically, educationally, culturally and professionally. For example, students could range from school leavers to mature-aged students, and anywhere in between. The entry standard, even of those who studied computing before, varies from year to year. They all share one common factor, though. Unfortunately, their prior educational experiences in the computing discipline may have done anything but prepare them appropriately for professional learning and practice. This is certainly the reason why programming courses are regarded by students as difficult and have high dropout rates [10].

Teachers now face the daunting task of somehow making sense - to this conglomeration of non-computing clientele - of the concepts such as initializing a sum, counting variables, using a correct looping strategy for the given problems [11]. As a result, often the instruction has primarily focused on programming *knowledge* [italised for emphasis] and it has been presented in a similar manner to the traditional curriculum, where the instructional materials consisted of several small exercises and assignments to be completed by students individually. In other words, class-work is typically simplified to enable students to engage in manageable chunks of work more focused on completion of the course rather than ensuring acquisition of programming *skills* [italised for emphasis]. Periodical assignments come neatly packaged with a well-specified set of requirements to be implemented. In the process, the goal of teaching ICT to prepare learners to achieve a holistic view of the computing problems and provide solutions is deplorably lost. In this context, we raise some questions relating to the goal of teaching ICT: Is it to acquire knowledge or skills? Is it to help students pass the exam or survive in the industry? Quite often teachers take cover under beautifully designed and presented theoretical frameworks for their lack of clarity on the goals of teaching ICT. ACM & IEEE-CS Joint Task Force on Computing Curricula [12] in their reviews since1991 has maintained that programming as 'activities that surround the description, development and effective implementation of algorithmic solutions to well- specified problems'. Also the emphasis on 'well-specified' problems becomes problematic when the focus shifts from '*developing programmes in the class-room*' to '*developing systems in the real world scenarios*' [13]. By this, students seemed to be expected to learn strategies implicitly by seeing examples and solving 'neatly graded' problems in class that can automatically be transferred to the industry. While this being so, another question very frequently asked is whether assessment in ICT, particularly that of computer programming can be a simple 'written language exercise' [14]. The above issues may find clarification by examining the dialectic programming dichotomies as a teaching pedagogy, and are discussed in the next section.

3.Competing Dichotomies for Computer Programming

It is generally accepted that it takes about ten years of experience to turn a student programmer into an expert programmer [9,15]. While there can be a lot of debate as to the definition of an 'expert programmer' the following section outlines three dialectic programming dichotomies that influence the teaching of computer programming.

3.1. Knowledge Versus Application

Studies show that there are positive correlations between the knowledge students' gain from instructional materials and the skill they develop by applying it for solving problems [16,17,18]. Obviously, programming ability must rest on a foundation of knowledge; it is, however, possible to distinguish programming knowledge from programming strategies. Knowledge, as it is understood, involves the declarative nature (syntax and semantics) of a programming language, while strategies describe how programming knowledge is applied [19]. Knowledge is only part of the picture; programming strategies involve the application of programming knowledge to solve a problem. Soloway [20] describes programming strategies as plans and Wallingford [21] views them as patterns or algorithms. A strategy, of course, is to be able to incorporate the plans, patters and algorithms into a single solution. Whalley et al [8] therefore, feel that teaching should reach beyond a focus on syntax, and target programming strategies. Robins et al [10] suggest that the key to beginner-programmers to becoming expert programmers lie in learning programming *strategies* rather than merely acquiring programming *knowledge*.

3.2. Comprehension Versus Generation

Another distinction is found between programme-comprehension (the ability to read and understand the outcomes of an existing piece of code) and programme-generation (the ability to create a piece of code that achieves certain outcomes). Whalley et al [8] contend that "a vital [initial] step toward being able to write programmes is the capacity to read a piece of code and describe it" (p. 249). It means that a student learning programming must be able to comprehend a solution (and the knowledge and strategies within it) before they can generate a solution at the same level of difficulty or rigour. According to Brooks [22], experts and beginner programmers can be distinguished by how they undertake comprehension. Again, with the ability to comprehend code comes the ability to reuse the pieces of code. It is widely recognized that practicing reuse does not happen automatically [23,24,25,26].

3.3. Object-Oriented Versus Procedure-Oriented

According to ACM & IEEE-CS Joint Task Force on Computing Curricula [12], object-oriented programming emphasizes the principles of design from the very beginning. Object-oriented approach has been regarded as ‘natural, easy to use and powerful’ in the sense that objects are natural features of problem domains, and are represented as explicit entities in the programming domain, so the mapping between domains is simple and should support and facilitate object-oriented design/programming. However, Detienne [27], Muller et al. [28] and Mittermeir et al. [29] do not support this position. They argue that identifying objects is not an easy process, that objects identified in the problem domain are not necessarily useful in the program domain, that the mapping between domains is not straightforward, and further that students need to construct a model of the procedural aspects of a solution in order to properly design objects/classes. While the literature on expert programmers is more supportive of the naturalness and ease of object-oriented design, it also shows that expert object-oriented programmers use both object-oriented and procedural views of the programming domains, and switch between them as necessary [27]. Similarly Rist [30] describes the relationship between plans (a fundamental unit of program design) and objects as ‘‘orthogonal’’ [30]. Yet the proponents of the objects-first strategy begin immediately with the notion of objects, classes, methods, constructors, and inheritance, and then go on to introduce concepts of types, variables, values etc. [31]. Having to assimilate all these details and to gradually build up new knowledge comprises one of the biggest sources of difficulties for student or beginner-programmers.

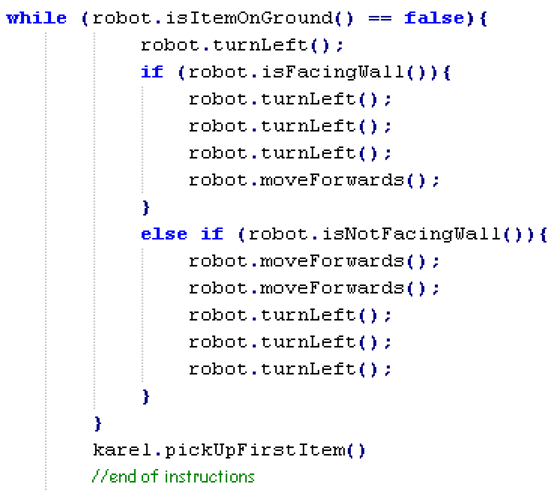
3.4. Functional Versus Imperative

The functional programming paradigm supports a pure functional approach to problem solving. Functional programming is a form of declarative programming. It involves composing the problem as a set of functions to be executed. Students need to define carefully the input to each function, and what each function returns. In contrast, in an imperative approach to teaching, students develop a piece of code that describes in exact detail the steps that the computer must take to accomplish the goal. This is often referred to as algorithmic programming. Most mainstream languages, including object-oriented programming (OOP) languages such as C#, Visual Basic, C++, and Java –, were designed to primarily support imperative (procedural) programming. These two strategies have been used for a fairly long period of time. The *functional* strategy initially places emphasis on *functions* leaving the presentation of *state* for later, whereas in the *imperative* strategy the emphasis is first given to the *state* and then the concept of *functions* is presented.

The four competing dichotomous approaches, discussed above, seem to be conflicting. Yet, teaching computer programming effectively may require an integrated approach that combines all the competing dichotomous approaches in proportions appropriate to the class situation. The choice of proportion may depend on the class composition and dynamics of learning.

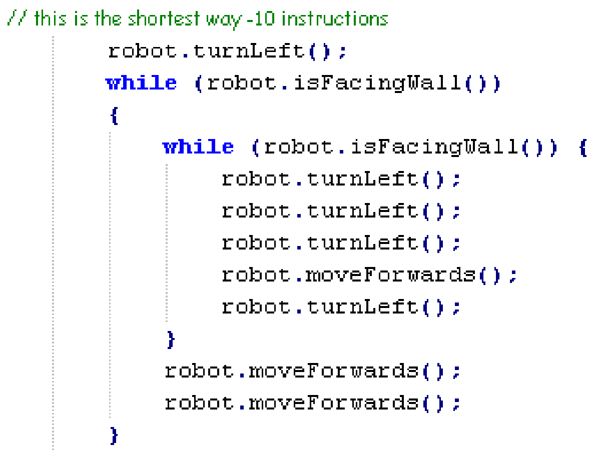
4. A Practical Approach to Teaching Programming

In addition to being able to produce compilable, executable programs that are correct and in the appropriate form, the students of computing should learn the process of solving discipline-specific problems irrespective of the particular programming paradigm. When faced with the crisis of computing student performance, the first step McCracken et al. [32] proposed was to abstract the problem from its description. Abstraction requires students to be well grounded in the idea of abstraction, starting from sub-algorithmic level.



```
while (robot.isItemOnGround() == false){
        robot.turnLeft();
        if (robot.isFacingWall()){
            robot.turnLeft();
            robot.turnLeft();
            robot.turnLeft();
            robot.moveForwards();
        }
        else if (robot.isNotFacingWall()){
            robot.moveForwards();
            robot.moveForwards();
            robot.turnLeft();
            robot.turnLeft();
            robot.turnLeft();
        }
    }
    karel.pickUpFirstItem()
    //end of instructions
```

Figure 1. Abutment



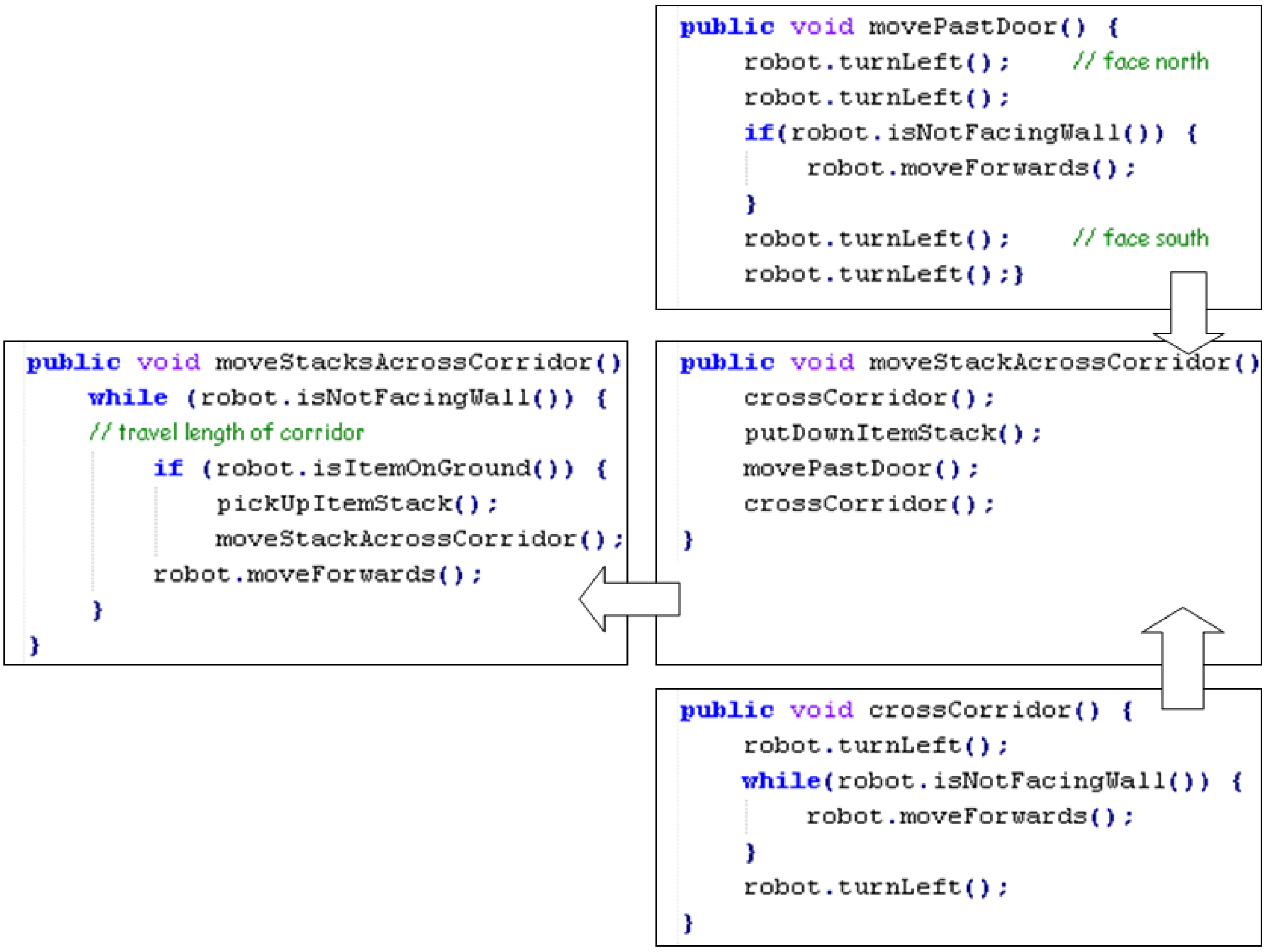
```
// this is the shortest way -10 instructions
    robot.turnLeft();
    while (robot.isFacingWall())
    {
        while (robot.isFacingWall()) {
            robot.turnLeft();
            robot.turnLeft();
            robot.turnLeft();
            robot.moveForwards();
            robot.turnLeft();
        }
        robot.moveForwards();
        robot.moveForwards();
    }
```

Figure 2. Nesting

At early stages of the course, relatively less detailed coding is required of the students but availability of a good selection of reusable classes and templates is essential. Eventually, algorithm analysis can become a springboard for principles of designing containers, as well as classic sorts and searches. By the end of the course, the students should have learnt that the default programming strategy is to reuse but they should have the concepts to start from scratch, if need be. Better still, since they will be thoroughly schooled in reuse, if they do code from scratch, they are more likely to think in terms of good abstractions that can be reused. For example, the method calls used in Figure 1 such as *turnLeft()* or *moveForwards()* are reused quite a few number of times in different sequences in order to meet specific needs which require of the students the skill of abstraction at sub-algorithmic level. de Raadt et al. [33] call this strategy 'abutment' which is calling one method after another in the correct sequence that will solve the problem. This also involves the functional approach where the problem is composed as a set of functions to be executed. While developing those functions, one needs to define carefully the input to each function, and what each function returns. This is the ability teachers need to keep as the objective of their teaching at this level.

The scope and importance of this strategy may be dependent on the design approach adopted in the problem-solving process. However, the functional decomposition of a structured program often requires further decomposition. In teaching computer programming, abstraction is not only a required skill in designing the classes needed, but also in factorization of methods out of others that are already in the design. For example, 'nesting' or placing one action sequence inside another is another form of abstraction (Figure 2). In this strategy, the student must be able to take the sub-solutions and put them back together to generate the solution to the problem. This step may involve creating an algorithm that controls the sequence of events.

The next level (Figure 3) of abstraction requires the students to be able to decide on an implementation strategy for individual classes, procedures, functions, or modules, as well as on appropriate language constructs. Although the solution should be correct and in the appropriate form that produces the right output, the emphasis, however, is that it should also be modularized, generalized, and conforms to standards. The focus for this strategy is the division of code into methods, and method signatures and names.



```
public void movePastDoor() {
    robot.turnLeft();      // face north
    robot.turnLeft();
    if(robot.isNotFacingWall()) {
        robot.moveForwards();
    }
    robot.turnLeft();      // face south
    robot.turnLeft();}
```

```
public void moveStacksAcrossCorridor()
    while (robot.isNotFacingWall()) {
    // travel length of corridor
        if (robot.isItemOnGround()) {
            pickUpItemStack();
            moveStackAcrossCorridor();
        robot.moveForwards();
    }
}
```

```
public void moveStackAcrossCorridor()
    crossCorridor();
    putDownItemStack();
    movePastDoor();
    crossCorridor();
}
```

```
public void crossCorridor() {
    robot.turnLeft();
    while(robot.isNotFacingWall()) {
        robot.moveForwards();
    }
    robot.turnLeft();
}
```

Figure 3. Modularisation

Another aspect that needs mentioning here is that software development often adopts one or several architectural patterns as strategies for system organisation. Expert programmers use these patterns purposefully. They in fact often use them informally and even nearly unconsciously. Good teaching needs to close the gap between the useful abstractions (constructs and patterns) of system design and the current models, notations and tools [34]. Teachers need to identify useful patterns clearly and teach them explicitly by giving examples, comparing them, and evaluating their utility in various settings allowing students to develop a repertoire of useful techniques that go beyond the curricular limitations. Let alone the issues that might crop up when larger systems are to be developed, several previous studies find weaknesses in teaching a computer programming course to beginners where *strategies* were not taught explicitly [11].

5. Discussion and Implications

Dede [35] observes that no educational ICT is universally good, and the best way forward is to take instrumental approach and analyse the curriculum, teachers and students in order to select appropriate tools, applications, media, and environments. In the context of teaching IT programming, scholars [36,37,38] have shown that explicit instruction strategies can be very powerful especially with regard to teaching programming. Recent studies [27,28,29] have focused on teaching patterns in an attempt to represent sub-algorithmic strategies. The four competing dichotomous approaches may have to be used in appropriate combinations that match the requirements of the class composition and student dynamics. It should be noted that these combinations may change as per the students' needs in each unique situation.

Some ways in which programming strategies could be incorporated in assignments and examinations is suggested below for consideration of ICT teaching community.

- Encouraging students to use particular strategies when generating solutions for assignments
- Awarding credit for application of strategies in assignment marking criteria
- Using problems that focus on programming strategies as part of the final examination
- Awarding credit for applying strategies in assessments was also done to encourage students to value this component of programming and devote more effort to learning it.

Academic and industry skill standards are needs closure integration in their design, development and dissemination. Teachers cannot undermine the relationship between academic and industry skill standards and the need to strive in order to reach a consensus in several central areas for better coordination between academic and technical standards. Workplace applications offered by the academic skills were rarely explicit. Industry skill standards included academic standards as an abstract list of skills would remain unconnected to their use in the workplace. Despite consensus that standards should be set at a high level, most academic standards offered no absolute normative benchmarks against which to measure student performance and were set by educators based on their judgment about what students should know in respective courses. The academic component of the industry skill standards call for skills that could be achieved well short of high school graduation. The most significant area of overlap between the two sets of standards was their use of process-oriented skills, which needs emphasis at the tertiary level teaching of computer programming.

Teaching programming with a clear emphasis on different strategies develops in beginner students several ways and means of problem solving. In traditional programming paradigm, teaching programming strategies is analogous to teaching programming design. Traditional computer science programmes place emphasis teaching programming design and analysis in the upper level courses such as system analysis and software development [40]. Students in an introductory programming course usually have limited exposure to program design.

6.Conclusion

The framework for strategies-oriented teaching of computer programming can be a clear process with very specific steps. Firstly, teaching starts off with the introduction of language features. This is followed, as a second step, by the discussion of a number of algorithmic solutions with the explicit objective of developing critical thinking skills. Logically the third step requires the students to debug the entire programme. Debugging is particularly important if the entire programme is actually developed in segments of classroom tasks.

As a project management approach to problem solving, problems are normally broken down into mini-tasks. As a teaching methodology, strategies can be explicitly applied to those mini-tasks. Each of these strategies can be illustrated in flowchart which should be achievable with no more than 10 lines of code. Tasks can also be formatted as a flow of checklists; each task should be solved sequentially.

The act of programming is a process, and the output of this process should be a working program. If the resultant programme does not work, then the process has not been successful and it is very hard to evaluate the process. It is simpler to give computing students the environment in which they can produce the programme and then only mark working programmes. Once the students know that the only way to pass the assessments is to learn how to programme, they develop a real interest in learning how to programme. This motivation makes it much easier to teach these students. But there is no substitute to hard work and trying it all over again. In Greek legend, Sisyphus, a king in ancient Greece who offended Zeus was given a punishment to roll a huge boulder to the top of a steep hill; each time the boulder neared the top it rolled back down and Sisyphus was forced to start again. We believe that the beginner-programmers need to have the tenacity of Sisyphus combined with understanding of modern business requirements that could help them succeed as computer programmers.

References

[1] Lahtinen, E., AlaMutka, K., & Järvinen, H. (2005). A Study of the Difficulties of Novice Programmers. *Proceedings of the ITiCSE'05 Conference,* June 27-29, Monte de Caparica, Portugal.

[2] Milne, I., & Rowe, G. (2002). Difficulties in Learning and Teaching Programming-Views of Students and Tutors. *Education and Information Technologies, 7*(1), 55-66.

[3] Pea, R. (1986). Language independent conceptual bugs in novice programming. *Educational Computing Research, 2*(1), 25-36.

[4] Saeli, M., Perrenet, J., Jochems, W. M. G., & Zwaneveld, B. (2011). Teaching Programming in Secondary School: A Pedagogical Content Knowledge Perspective. *Informatics in Education, 10*(1), 73-88.

[5] Kaczmarczyk, L., East, J. P., Petrick, E. R., & Herman, G. L. (2010). Identifying Student Misconceptions of Programming. *SIGCSE'10,* March 10-13, Milwaukee, Wisconsin, USA.

[6] de Raadt, M., Toleman, M., & Watson, R. (2006): Chick Sexing and Novice Programmers: Explicit Instruction of Problem Solving Strategies. *Australian Computer Science Communications, 28*(5):55-62.

[7] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004): A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin, 36*(4):119-150.

[8] Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robins, P., Kumar, P. K. A., & Prasad, C. (2006): An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Proceedings of the Eighth Australasian Computing Education Conference (ACE2006),* Hobart, Australia 52, 243-252.

[9] Bennedsen, J., & Caspersen, M. E. (2008). Optimists have more fun, but do they learn better? On the influence of emotional and social factors on learning introductory computer science. *Computer Science Education, 18*(1), 1-16.

[10] Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education, 13*(2):137-173.

[11] de Raadt, M., Toleman, M., & Watson, R. (2004): Training strategic problem solvers. *ACM SIGCSE Bulletin, 36*(2):48 - 51.

[12] ACM & IEEE-CS Joint Task Force on Computing Curricula 2001 (2001). Computing Curricula 2001, Ironman Draft. *Association for Computing Machinery and the Computer Society of the Institute of Electrical and Electronics Engineers.* Available: http://www.acm.org/sigcse/cc2001 [2001, 5/16/01].

[13] Clear, T (2001). "Programming in the Large" and the need for professional discrimination. *SIGCSE Bull. 33*, 4, 9-10.

[14] Coburn, D., & Miller, A. (2004). Assessment in Technology is not a Written Language Exercise. *SET, 44-48.*

[15] Winslow, L.E. (1996). Programming pedagogy – A psychological Overview. *SIGCSE Bulletin, 28*, 17-22.

[16] Chi, M., Bassock, M., Lewis, M. Reimann, P. and Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science, 13*, 145-182.

[17] Pirolli, P., & Recker, M. (1994). Learning strategies and transfer in the domain of programming. Cognition and Instruction, 12, 235-275.

[18] Gerdes, A., Jeuring, J. T., and Heeren, B.J, (2010). Using strategies for assessment of programming exercises. In *Proceedings of the 41st ACM technical symposium on Computer science education* (SIGCSE '10). ACM, New York, NY, USA, 441-445.

[19] Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies, 39*(2):237-267.

[20] Soloway, E. (1985): From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research, 1*(2):157-172.

[21] Wallingford, E. (2007) The Elementary Patterns Home Page, http://cns2.uni.edu/~wallingf/patterns/elementary/. Accessed 18th April 2011.

[22] Brooks, R. E. (1983): Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies, 18*:543-554.

[23] Auer, K. (1995). Smalltalk training: As innovative as the environment. *Comm. ACM, 38*(10), 115-117.

[24] Berg, W., Cline, M., & Girou, M. (1995). Lessons learned from the OS/400 OO Project. *Comm. ACM, 38*(10), 54-64.

[25] Fayad, M. E., & Tsai, Wei-Tek (1995). Object-oriented experiences. *Comm. ACM, 38*(10), 51-53.

[26] Frakes, W. B., & Fox, C. J. (1995). Sixteen questions about software reuse. *Comm. ACM. 38*(6) 75-87,112.

[27] Detienne, F. (1990). Expert programming knowledge: A schema based approach. In J.M. Hoc,T.R.G. Green, R. Samurc¸ay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 205-222). London: Academic Press.

[28] Muller, O., Haberman, B., & Ginat, D. (2007): Pattern-Oriented Instruction and its Influence on Problem Decomposition and Solution Construction. *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2007),* Dundee, Scotland.

[29] Mittermeir, R., Syslo, M., Benaya, T., & Zur, E. (2008). Understanding Object Oriented Programming Concepts in an Advanced Programming Course. *Informatics Education - Supporting Computational Thinking* (Vol. 5090, pp. 161-170): Springer Berlin / Heidelberg.

[30] Rist, R.S. (1995). Program structure and design. *Cognitive Science, 19*, 507-562.

[31] Chang, C., Denning, P. J., Cross, J. H., Engel, G., Roberts, E., & Shackelford, R. (2001). Computing curricula 2001. *ACM Journal of Educational Resources in Computing, 1*(3), 240.

[32] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bull*, 33, 4 (December 2001), 125-180.

[33] de Raadt, M., Watson, R & Toleman, M. (2009). Teaching and assessing programming strategies explicitly, Proceedings of the Eleventh Australasian Conference on Computing Education. In Margaret Hamilton and Tony Clear (Eds.), Vol. 95. *Australian Computer Society*., Darlinghurst, Australia, Australia, 45-54.

[34] Shaw, M., and Garlan, D., (1996). *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[35] Dede, C. (2008). Theoretical perspectives of influencing the use of information technology in teaching and learning. In J. Voogt and G. Knezek, (eds.). *International Handbook of Information Technology in Education.* New York: Springer.

[36] Biederman, I., & Shiffrar, M. M. (1987): Sexing Day-Old Chicks: A Case Study and Expert Systems Analysis of a Difficult Perceptual-Learning Task. *Journal of Experimental Psychology: Learning, Memory and Cognition, 13*(4):640-645.

[37] Reber, A. S. (1993). *Implicit Learning and Tacit Knowledge.* New York, USA: Oxford University Press.

[38] de Raadt, M., Toleman, M., & Watson, R. (2007): Incorporating Programming Strategies Explicitly into Curricula. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research* (Koli Calling 2007), Koli, Finland, 53-64.

[39] Porter, R., & Calder, P. (2003). A Pattern-Based Problem-Solving Process for Novice Programmers. *Proceedings of the Fifth Australasian Computing Education Conference (ACE2003),* Adelaide, Australia 20:231-238, Conferences in Research and Practice in Information Technology.

[40] Ghafarian, A. (2001). Teaching design effectively in the introductory programming courses. *J. Comp. Sci. in Colleges*, vol. 16, no. 2, pp. 201-208.