PROCEEDINGS

of the Union of Scientists - Ruse

Book 5

Mathematics, Informatics and Physics

Volume 10, 2013

RUSE

The Ruse Branch of the Union of Scientists in Bulgaria was founded in 1956. Its first Chairman was Prof. Stoyan Petrov. He was followed by Prof. Trifon Georgiev, Prof. Kolyo Vasilev, Prof. Georgi Popov, Prof. Mityo Kanev, Assoc. Prof. Boris Borisov, Prof. Emil Marinov, Prof. Hristo Beloev. The individual members number nearly 300 recognized scientists from Ruse, organized in 13 scientific sections. There are several collective members too – organizations and companies from Ruse, known for their success in the field of science and higher education, or their applied research activities. The activities of the Union of Scientists – Ruse are numerous: scientific, educational and other humanitarian events directly related to hot issues in the development of Ruse region, including its infrastructure, environment, history and future development; commitment to the development of the scientific organizations in Ruse, the professional development and growth of the scientists and the protection of their individual rights.

The Union of Scientists – Ruse (US – Ruse) organizes publishing of scientific and popular informative literature, and since 1998 – the "Proceedings of the Union of Scientists- Ruse".

BOOK 5

"MATHEMATICS, INFORMATICS AND PHYSICS"

VOLUME 10

CONTENTS

BOOK 5

"MATHEMATICS, INFORMATICS AND PHYSICS"

VOLUME 10

This is the jubilee 10-th volume of book 5 Mathematics, Informatics and Physics. The beginning was in Spring, 2001, when the colleagues of the former section Mathematics and Physics decided to start publishing our own book of the Proceedings of the Union of Scientists – Ruse. The first volume included 24 papers. Through the years there have been authors not only from the Angel Kanchev University of Ruse but as well as from universities of Gabrovo, Varna, Veliko Tarnovo and abroad – Russia, Greece and USA.

Since the 6-th volume the preparation and publishing of the papers began to be done in English.

The new 10-th volume of book 5 Mathematics, Informatics and Physics includes papers in Mathematics, Informatics and Information Technologies, Physics and materials from the Scientific Conference 'Information Technologies in Education' (ITE), held at the University of Ruse in November 2012 in the frame of Project 2012-FNSE-02.

web: suruse.uni-ruse.bg

DIDACTIC AIMS AND PERSPECTIVES IN COMPUTER SCIENCE TEACHING

Galina Atanasova

Angel Kanchev University of Ruse

***Abstract**: The article discusses the scope of education in computer science and the didactic objectives on which the introductory courses on algorithms based on those of programming to be built. Attention is paid to the variety aspects of the material and the reasons how to be precisely structured its presentation. The paper proposes an approach of 4 levels for learning and development of skills for writing algorithms. It stresses on the importance students to acquire skills for writing correct and efficient algorithms to achieve success in programming.*

***Keywords:** Computer Science, Algorithm, Algorithm Teaching, Algorithm skills development.*

INTRODUCTION

Computer Science is the study of principles and practices that underpin an understanding and modelling of computation and of their application in the development of computer systems [1]. At its heart the notion of computational thinking lies: a mode of thought that goes well beyond software and hardware and that provides a framework within which to reason about systems and problems. This mode of thinking is supported and complemented by a substantial body of theoretical and practical knowledge and by a set of powerful techniques for analysing, modelling and solving problems.

Computer Science is deeply concerned with how computers and computer systems work and how they are programmed. Students studying computing gain insight into computational systems of all kinds, whether or not they include computers. It allows them to solve problems, design systems and understand the power and limits of human and machine intelligence. It is a skill that empowers and all students should be aware of and have some competence in. Furthermore, students who can think computationally are better able to conceptualise and understand computer-based technology and so are better equipped to function in modern society.

Computer Science is a subject, where invention and resourcefulness are encouraged. Students are expected to apply the academic principles they have learned to understanding real-world systems and creating purposeful artefacts. This combination of principles, practice and invention makes Computer Science an extraordinarily useful and an intensely creative subject, suffused with excitement, both visceral (“it works!”) and intellectual (“that is so beautiful”).

THE BASIC AIMS OF COMPUTER SCIENCE

Education enhances students’ lives as well as their life skills. It prepares young people for a world that doesn’t yet exist; involves technologies that have not yet been invented; presents technical and ethical challenges of which we are not yet aware.

To do this, education aspires primarily to teach disciplines with long-term value, rather than skills with short-term usefulness, although the latter are certainly useful. A “discipline” is characterised by:

- A body of knowledge, including widely-applicable ideas and concepts and a theoretical framework into which these ideas and concepts fit.

- A set of techniques and methods that may be applied in the solution of problems and in the advancement of knowledge.
- A way of thinking and working that provides a perspective on the world that is distinct from other disciplines.
- Longevity: a discipline does not “date” quickly, although the subject advances.
- Independence from specific technologies, especially those that have a short shelf-life.

Computer Science comprises disciplines with all these characteristics. It encompasses foundational principles, such as the theory of computation and widely applicable ideas and concepts, such as the use of relational models to capture structure in data [4]. It incorporates techniques and methods for solving problems and advancing knowledge, such as abstraction and logical reasoning, and a distinct way of thinking and working that sets it apart from other science areas (computational thinking). It has longevity (most of the ideas and concepts that were current 20 or more years ago are still applicable today) and every core principle can be taught or illustrated without relying on the use of a specific technology.

THE KEY DIDACTIC AIM

The key didactic aim is something that a student of Computer Science should be able to do and should know. In Computer Science the key processes focus upon computational thinking. Computational thinking is the process of recognising aspects of computation in the world that surrounds us and applying tools and techniques from computing to understand and reason about both natural and artificial systems and processes.

Computational thinking is something that people rather do than computers and includes the ability to think logically, algorithmically and at higher levels recursively and abstractly [2]. It is, however, a rather broad term. The rest of this paper draws out particular aspects of computational thinking that are particularly accessible to and important for young people at the universities.

A well-grounded student of Computer Science will also be proficient in other generic skills and processes including: thinking critically, reflecting on ones work and that of others, communicating effectively both orally and in writing, being a responsible user of computers and contributing actively to society.

ABSTRACTION: MODELLING, DECOMPOSING AND GENERALISING

A key challenge in computational thinking is the scale and complexity of the systems we study or build [2]. The main technique used to manage this complexity is abstraction. The process of abstraction takes many specific forms, such as modelling, decomposing and generalising. In each case, complexity is dealt with by hiding complicated details behind a simple abstraction or model of the situation. For example, the Ruse Bus Route map is a simple model of a complex reality — but it is a model that contains precisely the information necessary to plan a route from one station to another. A procedure to compute square roots hides a complicated implementation (iterative approximation to the root, handling special cases) behind a simple interface (give me a number and I will return its square root).

Modelling is the process of developing a representation of a real world issue, system or situation, that captures the aspects of the situation that are important for a particular purpose, while omitting everything else. Different purposes need different models. Example: a geographical map of the Bus Route is more appropriate for computing travel times than the well-known topological Bus Route map; a network of nodes and edges can be represented as a picture or as a table of numbers. A particular situation may need more

than one model. For example a web page has a structural model (headings, lists, paragraphs) and a style model (how a heading is displayed, how lists are displayed). A browser combines information from both models as it renders the web page.

A problem can often be solved by decomposing it into sub-problems, solving them and composing the solutions together to solve the original problem. For example “Make breakfast” can be broken down into “Make toast; make tea; boil egg”. Each of these in turn can be decomposed, so the process is naturally recursive. The organisation of data can also be decomposed. For example, the data representing the population of a country can be decomposed into entities such as individuals, occupations, places of residence, etc.

Sometimes this top-down approach is the way in which the solution is developed but it can also be a helpful way of understanding a solution regardless how it was developed in the first place. For example, an architectural block diagram showing the major components of a computer system (e.g. a client, a server, and a network) and how they communicate with each other, can be a very helpful way of understanding that system.

Complexity is often avoided by generalising specific examples, to make explicit what is shared between the examples and what is different about them. For example, having written a procedure to draw a rectangle of sizes 2 and 4 and another to draw a rectangle of sizes 3 and 5, one might generalise to a procedure to draw a rectangle of any sizes N and M, and call that procedure with parameters 6 and 7 respectively. In this way much of the code used in different programs can be written once, debugged once, documented once, and (most important) understood once. A different example is the classification encouraged by object-oriented languages, whereby a parent class expresses the common features of an object, for example, the size or colour of a shape, while the sub-classes express the distinct features (a square and a triangle, perhaps). This process may be called generalisation. It is the process of recognising these common patterns and using them to control complexity by sharing common features.

PROGRAMMING

Computer Science is more than programming, but programming is an absolutely central process for Computer Science. In an educational context, programming encourages creativity, logical thought, precision and problem-solving and helps foster the personal, learning and thinking skills required in the modern university curriculum. Programming gives concrete, tangible form to the idea of “abstraction” and repeatedly shows how useful it is [5].

Every student should have repeated opportunities to design, write, run and debug executable programs [3]. What an “executable program” means can vary widely, depending on the level of the students’ skills and the amount of time dedicated for. In some cases the ability to understand and explain a program is much more important than the ability to produce working but incomprehensible code [4]. Depending on level of their skills, students should be able to:

1. Design and write programs that include
- Sequencing: doing one step after another.
- Selection (if-then-else): doing either one thing or another.
- Repetition (Iterative loops or recursion).
- Language constructs that support abstraction: wrapping up a computation in a named abstraction, so that it can be re-used. (The most common form of abstraction is the notion of a “procedure” or “function” with parameters.).
- Some form of interaction with the program’s environment such as input/output, or event-based programming.

2. Find and correct errors in their code.

3. Reflect thoughtfully on their program, including assessing its correctness and fitness for purpose; understanding its efficiency and describing the system to others.

Effective use of the abstraction mechanisms supported by programming languages (functions, procedures, classes, and so on) is central to managing the complexity of large programs. For example, a procedure supports abstraction by hiding the complex details of an implementation behind a simple interface. These abstractions may be deeply nested, layer upon layer. Example: a procedure to draw a rectangle calls a procedure to draw a line; a procedure to draw a line calls a procedure to paint a pixel; the procedure to paint a pixel calls a procedure to calculate a memory address from an (x, y) pixel coordinate. As well as using procedures and libraries built by others, students should become proficient in creating new abstractions of their own. A typical process is:

- Recognise that one is writing more or less the same code repeatedly. Example: draw a rectangle of sizes 2 and 4; draw a rectangle of sizes 3 and 5.
- Designing a procedure that generalises these instances. Example: draw a rectangle of sizes N and M.
- Replace the instances with calls to the procedure. At a higher level, recognising a standard “design pattern”, and re-using existing solutions, is a key process. For example: simple data structures, such as variables, records, arrays, lists, trees, hash tables.
- Higher level design patterns: divide and conquer, sorting, searching, backtracking, recursion.

Students also must have abilities for debugging, testing, and reasoning about programs. When a programmed system goes wrong, they have to answer the question “How can I fix it?” Computers can appear so opaque that fault-finding degenerates into a demoralising process of trying randomly generated “solutions” until something works. Programming gives students the opportunity to develop a systematic approach to detecting, diagnosing and correcting faults, and to develop debugging skills, including:

- Reading and understanding documentation.
- Explaining how code works or might not work.
- Manually executing code, perhaps using pencil and paper.
- Isolating or localising faults by adding tracing.
- Adding comments to make code more human readable.
- Adding error checking code to check internal consistency and logic.
- Finding the code that causes an error and correcting it.
- Choosing test cases and constructing tests.

A SYSTEMATIC APPROACH IN LEARNING COMPUTER SCIENCE

At the university level of education we need to ensure the process with a systematic approach in knowledge acquiring. Data structures and algorithms are important foundation topics in Computer science education. Students deal with algorithms in many computer science courses and so they must be equipped with solid skills in algorithms [6]. We suggest a progressive building on approach for algorithm teaching, divided in four levels, presenting below. Each level upgrades the knowledge acquired on the previous one.

Level 1 introduces the algorithm concept by following explanations:

- Presenting the algorithms as they are sets of instructions for achieving goals, made up of pre-defined steps. Example: a recipe for making fried eggs.
- Algorithms can be represented in simple formats narrative text.
- They can describe everyday activities and can be followed by humans and by computers.
- Computers need more precise instructions than humans do.

- Steps can be repeated and some steps can be made up of smaller steps.

Level 2 enriches the basic concepts from the previous level 1 with forming the following skills:

- Algorithms representation symbolically (flowcharts) or using instructions in a clearly defined language.
- Building algorithms that include selection (if) and repetition (loops).
- Algorithms decompositions into component parts (procedures), each of which itself contains an algorithm.

At this level 2 students should also be able to make:

- Algorithms should be stated without ambiguity and care and precision are necessary to avoid errors.
- Algorithms are developed according to a plan and then tested. Algorithms are corrected if they fail these tests.
- It can be easier to plan, test and correct parts of an algorithm separately.

The next level 3 comprises the described below concepts:

- An algorithm is a sequence of precise steps to solve a given problem.
- A single problem may be solved by several different algorithms.
- The choice of an algorithm to solve a problem is driven by what is required of the solution (such as code complexity, speed, amount of memory used, amount of data, the data source and the outputs required).
- The need for accuracy of both algorithm and data.

The last final level 4 turns significant milestones for the following students' skills:

- The choice of an algorithm should be influenced by the data structure and data values that need to be manipulated.
- Familiarity with several key algorithms (sorting and searching).
- The design of algorithms includes the ability to easily re-author, validate, test and correct the resulting code.
- Different algorithms may have different performance characteristics for the same task.

CONCLUSIONS AND FUTURE WORK

Future plans are to investigate the suggested four level approach in teaching algorithms for students that learn computer science. Currently, we use the pedagogy based on operational theories approach of learning development [7]. It will be in favor of education quality the two approaces to be compared. Of particular interest are the solving problems principles of suggested approach teaching pedagogy. However, this idea is still in the initial stages of development and much more work is needed. Firstly, further research is needed to refine the ideas regarding this teaching approach and how it can best be used to aid in the delivery of the introductory concepts of programming.

REFERENCES

[1] Byrne, P. & Lyons, G., The effect of student attributes on success in programming, Proceedings of the 6th annual conference on Innovation and technology in computer science education, 2001, pp. 49-52.

[2] Jeanette Wing, "Computational thinking", Communications of the ACM, March, 2006.

[3] Kurland, D., Pea, R., Clement, C. & Mawby, R., A Study of the development of programming ability and thinking skills in high school students. In Soloway & Spohrer: Studying the Novice Programmer, 1989, pp. 209-228.

[4] Perkins, D., Hanconck, C., Hobbs, R., Martin, F. & Simmons, R., Conditions of learning in novice programmers. In Soloway & Spohrer: Studying the Novice Programmer, 1989, pp. 261-279.

[5] Robins A, Rountree J, and Rountree N, Learning and teaching programming: A review and discussion, Computer Science Education - Volume 13 Issue 2, Routledge, Oxford England, 2003, pp: 137-172.

[6] Soloway, E. & Spohrer, J. Studying the Novice Programmer, Lawrence Erlbaum Associates, Hillsdale, New Jersey. 1989, p. 497.

[7] Леонтьев, А. Н. Деятельность, сознание, личность. Моссква, 1975.

CONTACT ADDRESS

Pr. Assist. Galina Atanasova
Department of Informatics and Information Technologies
Faculty of Natural Sciences and Education
Angel Kanchev University of Ruse
8 Studentska Str., 7017 Ruse,Bulgaria
Phone: (++359 82) 888 326
E-mail: gea@ami.uni-ruse.bg

ДИДАКТИЧЕСКИ ЦЕЛИ И ПЕРСПЕКТИВИ В ОБУЧЕНИЕТО ПО КОМПЮТЪРНИ НАУКИ

Галина Атанасова

Русенски университет "Ангел Кънчев"

Резюме: *Статията разглежда обхвата на обучението по компютърни науки, дидактическите цели, на базата на които да се изграждат въвеждащите курсове по алгоритми, които са в основата на тези по програмиране. Обърнато е внимание на многоаспектността на материала и причините, поради които трябва прецизно да се структурира поднасянето на материала. Статията предлага подход от 4 нива за усвояване на знания и изграждане на умения за съставяне на алгоритми. Подчертава важността студентите да усвоят умения за съставяне на коректни и ефективни алгоритми, за да постигнат успех в програмирането.*

Ключови думи: *Компютърни науки, Алгоритми, Обучението по алгоритми, Умения за съставяне на алгоритми*

ISSN 1314-3077