Teaching Software Designing Skills

Judith Gal-Ezer

The Open University of Israel

16, Klausner St. Tel-Aviv, Israel 61392

Phone: 972-3-6460218

Fax: 972-3-6460744

galezer@cs.openu.ac.il

Adital Zeldes

zeldes@netvision.net.il

Abstract

Research has revealed a significant gap between the thinking patterns and software design habits of students or recent computer science university graduates, and those of expert software developers. There may be several causes for this gap, one of which is undoubtedly the fact that throughout their studies students are mostly asked to write software programs of relatively limited scope, and are not usually required to work as part of a team constructing a complex program. In the present paper we will describe a study unit intended to be taught as part of a high school computer science curriculum, which emphasizes the software system aspects. In a study that accompanied the development and actual teaching of the unit, we checked, among other things, whether the students acquired skills for developing a complex software system.

Key words: software, high-school, curricula

Introduction

Software systems are actually very big and complex programs, often comprising hundreds of thousands of code lines. They perform many complex functions and handle a wide variety of inputs. Software systems are developed by teams, starting with the specification stage, through designing and implementation, all the way to marketing and distribution.

Software developers usually acquire their education at universities, colleges or similar institutions. A large portion of the programs written at school are, for obvious reasons, of relatively limited size. Students are required to write more complex software projects only two or three times during their studies - either independently or as members of a team. As a result, students do not always acquire sufficient programming skills for handling large systems.

And indeed, research does suggest a large disparity between the thinking habits of students and those of expert software developers [1]. Students prefer fast and simple writing, even if simplicity comes at the expense of maintenance capability, while experts know from experience that a program does not end when its writing has been completed, and therefore it is important to adhere to proper programming rules. In order for students to acquire correct programming habits, suitable for the development of large complex programs, they must be taught a system oriented approach and provided with an ability to cope with developing large software systems in the future.

In this paper we will focus on a study unit designed to teach software designing skills in high school. We will describe the unit and present the views of teachers and their students, about the study materials, the goals of the unit and the degree to which these goals have been attained. We will end the paper with conclusions and recommendations for the future.

Background

College and university computer science curricula usually begin with mandatory courses in mathematics (such as discrete mathematics, algebra, etc.), then an introductory course (in which the study material is usually implemented using one programming language) followed by a data structure course. A software engineering course is offered at a later stage, but even the first two courses already try to emphasize skills of correct designing and programming and teach sound habits for constructing software systems.

In recent years, as study program designers began to realize that computer science should be taught in high school, just like physics, chemistry or biology, high school curricula were prepared, largely similar in structure to university programs.

A computer science curriculum for high school was developed in Israel as early as the 1970's, by a committee nominated by the Ministry of Education. At the beginning of the 1990's, a new committee was set up and charged with the task of updating this program, termed hereafter the "old" program. The committee, examined the curriculum and written material and decided that it was inadequate. A new syllabus was then prepared, along with suitable study materials. A summary of the committee's work and the new curriculum are given in [2].

As part of the new curriculum, a study unit called *Software Design* was developed, intended for students who have completed the *Fundamentals 1, 2* units, and wish to expand and enhance their knowledge of computer science. The unit comprises 90 hours. Its development began in 1992, and in 1994 it was first tested in the classroom. In 1998 the unit became a mandatory part of the study curriculum.

One of the missions of the development team was to prepare written material, to prevent computer science teachers whose education is based on no more than a few programming courses, from emphasizing the syntax of the programming language, while neglecting the overall view. Focusing on syntax and specific examples does not encourage students to identify and use more complex structures. According to Linn and Clancy [3] programming students tend to organize their knowledge in categories of the programming language syntax, this practice hinders good use of knowledge, since no good connections are formed among the different parts of that knowledge. As a result, many students apply trial and error methods when trying to write a program for solving a problem. Experts however, keep their knowledge in conceptual structures that are broader than syntax. Thus, they can reuse patterns that do not depend on any particular programming language. The developers aimed at providing tools that will help students in organizing their knowledge the way experts do.

The unit's name, *Software Design* came to represent the gradual change of emphasis from the computer toward the people who use it. The unit stressed the notion that the task of those who create a new software program is to design the interaction between the computer and its users, and not merely the software itself. The designer and the user both create a world - they don't just put into the computer things that had previously existed elsewhere. The major part of designing is creating a consistent world based on a comprehensible structure or model. The tools for creating these worlds are usually based on object oriented models, in which the status of the various objects reflects the user's point of view. The task focuses on the definition and description of objects, on the features and actions operated by the user during interaction, rather than on the programming itself [6]. Usability and abstraction serve as guidelines throughout the unit, at the expense of actual programming. The unit's roots are thus nourished by a well founded theory of design, and do not rely upon any particular programming language, the subject are studies in a new light, through "the designer's spectacles".

In addition to computer science considerations, pedagogical considerations guided the developers. Studies have shown that good organization of knowledge helps people to remember and reuse it. Perkins [5] called this type of knowledge "generative knowledge". Generative knowledge fulfills the following three goals: Preserving knowledge for a long time, understanding knowledge, and using knowledge actively. Correct organization of knowledge helps the learner distinguish between the significant and the insignificant, and recognize similarities among different problems - thus promoting its internalization, understanding and application in various situations. Both the teacher and the text book play an important role in providing students with generative knowledge. The unit was designed with this in mind, aiming at providing tools for acquiring generative knowledge.

The *Software Design* Unit

Goals

The unit's main objectives are:

- Learning the principles of the system oriented approach including top down design, dividing a task into sub-tasks, team work and more.
- Experiencing the designing and construction of a small complex system from start to finish.

- Developing abstract thinking abilities, especially the ability to define new abstract data types from the basic types provided by the language.
- Becoming acquainted with known abstract data types such as list, stack, queue and binary tree, and using them to solve given problems.
- Acquiring the ability to analyze the efficiency of algorithms and the programs that implement them.
- Becoming acquainted with algorithms to more advance algorithmic problems like search and sort. Executing them on different data types, and comparing their efficiency.
- Acquiring the ability to choose suitable data types for the implementation of a solution to a given problem. Defining the types, and placing them at the user's disposal by writing suitable interfaces and implementing them in a programming language.

Contents

The unit introduces six main subjects, as detailed below.

The library unit: In-depth acquaintance with the library unit in Turbo-Pascal, elucidating the idea of modularity by building and using library units.
Data types: Acquaintance with the concept "abstract data type"; learning the definition, representation and implementation stages of abstract data types, and practicing extensively; handling exceptions when defining interfaces.
Stack: Acquaintance with the abstract data type "stack" and its various applications.
Efficiency: Understanding the term "efficiency"; emphasis is placed on the fact that some problems cannot be solved in reasonable time, even by very fast computers; acquaintance with the terms "input length" and "basic step"; understanding that a good measure of time complexity is the number of times an algorithm performs a basic step as dependent on the input length; using "the worst case" as the main measure for evaluating an algorithm's run time complexity; acquaintance with the term "Order of magnitude" (big O); acquaintance with families of orders of magnitude: Logarithmic, linear, square and exponential; understanding the difference between improving by a constant and improving by an order of magnitude; the ability to perform a basic analysis of an algorithm's run time complexity.
List: Acquaintance with the data type "list"; internalizing the principle of information hiding through acquaintance with various implementations of the same interface; acquaintance with dynamic memory allocation; implementing an abstract data type by using an existing data type, such as stack and queue that are implemented through a list.
Binary tree: Acquaintance with the data type "binary tree" and its various applications; practicing the application of recursive routines and their evaluation; acquaintance with a binary search tree and its applications.

Teaching aids

The unit is accompanied by several teaching aids:

Text Book

The text book *Software Design* [7] is meant to assist both teachers and students. The book covers the subjects mentioned above and offers a variety of exercises. Teachers can instruct the students to read topics in the book, either before or after teaching them in class, and may even use the text in class, as needed. The book is divided into six chapters, and each chapter deals with one of the six topics taught in the unit. These chapters are preceded by an introductory chapter presenting the general background for the *Software Design* study unit and preparing the students for the task that lies ahead by exposing them to the necessary basic concepts.

Teacher's guide
The teacher's guide [8] is structured in accordance with the unit's chapters. For each chapter, the guide includes the following information: Required teaching time, goals and objectives, a teaching process proposal, points of emphasis and comments, helpful didactic aids, concepts and key words, exercises and their solutions.

Case study
The case study is a project that accompanies the unit and develops as the unit progresses. The case study's purpose is to apply the principles learned in the unit while using the ideas of modular writing, abstraction and hiding, functional data structuring, and the writing of well defined user-machine interfaces.

The main goals of the case study are presenting a system consisting of several modules; combining the study materials into one full, complete picture; practicing the stages of specifying, designing and refining in the development of software systems; giving the students tools to reuse existing modules and to improve them, as needed.

The case study book [9] contains assignments for students and instructions for teachers. At present it offers two alternative tasks. Each teacher may choose the one more appropriate for his or her class, or propose a different task altogether. The case study is accompanied by a diskette containing all the stages required in the process of development.

Exercises and solutions
Theoretical questions integrated in the text book help guiding discussions and channel the study material. Exercises given in the text, intend to guide the students and steer class discussions. Homework exercises are also given at the end of each chapter. In addition, exercises and solutions on a diskette are distributed for the teacher's use, as well as printed exercises and solutions.

PowerPoint presentations
A collection of PowerPoint presentations for visual clarification of complex subjects are also provided.

The Research

Goals
The study that accompanied the development of the unit and its initial implementation was intended to check whether the goals set by the developers were being achieved, and whether the unit was suitable for its target population. We decided to focus on a limited number of aspects in order to draw conclusions and make final adjustments.

Two of the topics examined by the study were teachers' and students attitudes towards the unit and its contents, central ideas and teaching aids, and tools and skills acquired by the students for solving programming problems.

Participants

The study was conducted in high schools preparing their students for 5 unit matriculation examinations in computer science. The participants were:

Approximately 140 students who took the *Software Design* unit in the 1996-7 school year, as part of their study program.
Approximately 10 teachers who taught the unit during the same year.
Approximately 30 teachers who attended in-service training courses.

Tools

Follow up questionnaires for teachers

Three follow up questionnaires were administered to the teachers. Each questionnaire focused on the evaluation of some of the unit's chapters, and was administered soon after those chapters had been taught. Each questionnaire included an evaluation of the study materials: Scope, difficulty, quantity and quality of examples and exercises; the contribution of the teaching aids; a comparison between the contents of the new unit and a similar unit of the old program; critique of the unit's contents; the students' interest and participation; difficulties in conveying the study material; comments and suggestions. The first questionnaire also included details about classes and students.

Attitude questionnaires for teachers

The attitude questionnaire included 22 statements related to several areas pertaining to the unit and its contents: Pre-designing solutions, information hiding, the computer's efficiency and limitations, handling exceptions, abstract data types and the case study. The questionnaire also included statements regarding the target population, combining theory with practical work, and individual work as opposed to team work. The questionnaire was administered to teachers who attended in-service courses, and to experienced teachers who were already teaching the unit. The teachers were asked to indicate their degree of agreement with the statements on a scale of 0-4 (0 = haven't formed a clear opinion; 1 = definitely opposed; 2 = opposed; 3 = agree; 4 = definitely agree).

Attitude and concept questionnaires for students

This questionnaire, administered to students at both the beginning and end of the course, included the following topics: Personal details, attitudes (very similar to the teachers' attitude questionnaire), and knowledge of concepts. The concepts included both concepts that were included in the unit's prerequisites and concepts learned in the unit itself. The aim was to check the students' prior knowledge and their progress during the year. The final questionnaire also included the following topics: Evaluation of the unit's contents and text book (similar to the questions in the teachers' follow-up questionnaires) and definitions of the unit's basic concepts. These definitions were meant to ascertain whether the students understand the unit's fundamental concepts, such as abstract data types, implementation, module, interface, library unit. The questionnaire also provided some empty space for comments and suggestions.

Interviews with students
Piaget [4] has described the child as a scientist trying to understand the world, and true learning as the structuring of ideas, rather than memorizing information. True knowledge must be built and processed, and therefore cannot be a perfect mirror image of what has been learned. In the interviews the students were presented with a new problem which they had not encountered previously - in order to test the depth and significance of the learning process, and find out whether the unit had provided the students with tools for handling new problems. The problem presented to the students was designing a solution for a programming problem from the paper by Linn and Clancy [3]. In addition, the interviews included questions that examine the understanding of tools and concepts taught in the unit.
Interviews were conducted with 18 students from 5 different schools, all of whom had taken the "Software Design" study unit in 1997. Seven of the interviewees from 2 different schools were interviewed twice, at both the beginning and end of the course.

Findings

As mentioned before the research focused on two main questions. Below we describe the main findings as drawn from the questionnaires and interviews.

Attitudes of teachers and students

The unit, its contents, and teaching aids
The teachers emphasized the importance of the unit's topics and their considerable contribution to students interested in enhancing their knowledge of computer science. The teachers were very pleased to have a text book structured according to the mandatory program - which the old program did not provide. With the book at their disposal the teachers no longer needed to gather the study materials on their own. The text book played a very significant role. It was used extensively both in class and at home. The teachers used the examples and exercises presented in the book and guided the students to use them as well.

The teacher's guide proved very useful to all teachers, both new and experienced. The teachers utilized suggestions for teaching, and for planning the time required for each chapter. They also used the additional exercises and solutions extensively. The "Emphases and Comments" section helped the teachers anticipate difficult topics, and suggested ways for handling or preventing the problems.

The PowerPoint presentations served as an excellent means of illustrating and clarifying problematic issues, and were applauded by the teachers. This attractive, dynamic tool solved many problems for the teachers through visual presentation at difficult stages of the teaching process. The use made of the presentations encouraged the teachers to adopt this tool and apply it both freely and creatively. Many teachers prepared additional presentations.

Both teachers and students did, however, complain that the volume of study material was too vast in relation to the overall number of hours. Some of this excessive load resulted from the case study included in the unit. It must be noticed that most of the complaints were raised by teachers who were teaching the unit for the first time.

Teachers also expressed some dissatisfaction regarding exercises and laboratory sessions. These complaints focused on four main issues: Actual programming was somewhat neglected in favor of algorithmic writing; the number of laboratory sessions was insufficient for the tasks at hand; an insufficient number of questions suitable for the unit; difficulties in algorithmic writing.

As a consequence, the development team provided teachers with additional exercises and solutions during the recent school year, and held in-service courses. In addition, a special chapter on algorithmic writing was prepared for the benefit of both teachers and students.

Central ideas of the unit
Experienced teachers who had taught the parallel unit in the old program expressed conflicting attitudes toward the new unit. Some teachers welcomed the unit's fundamental ideas and began to teach them gradually, while others were opposed to changes in the unit's contents. The objections were mainly directed at the shift in focus from actual programming to abstract thinking and design. The unit emphasized system design and data abstraction, using algorithms written in pseudo-code. Teachers found it hard to depart from traditional flow charts and decrease the use of Pascal programs.

It should be re-emphasized that practical programming does not play a central role in the unit. At the end of the year students and teachers realized that programming in itself is not the main goal. Teachers learned to appreciate the importance of designing that is independent of the work environment, despite the fact that students (as well as some teachers) need something tangible such as a running program, in order to feel that they have actually accomplished something.

Main topics of the unit
The principle of information hiding: Among students, improvement was observed as the year went by. The principle of information hiding, which wasn't quite clear to them at the beginning of the year, was well internalized as learning progressed. Teachers were more aware of the importance of the principle, and their support of information hiding became clearer all along.

Efficiency: The most important observation here is that students who had considered the computer to be omnipotent at the beginning of the year, gradually learned that it had certain limitations, requiring the responsible programmer to apply efficiency considerations.

Handling exceptions: Both teachers and students agreed right from the beginning that programs must provide answers for exceptions, and that this does not detract from their efficiency.

Integrating theory and practice: On one hand, it was surprising to find among students an awareness of the greater importance of theoretical knowledge when compared with practical knowledge. However, most students still felt they should be given more programming tasks during their studies. The great majority of teachers, on the other hand, thought that practical knowledge is more important than theoretical knowledge, and most of them were in favor of giving students many programming tasks. No significant difference was found in this respect between teachers who were actually teaching the unit and their colleagues who only attended the in-service training course. All teachers agreed in principle that students should experience designing and writing a complete software system.

Individual work as opposed to team work: Students for the most part preferred team work over individual work. This preference grew as the year went by. Teachers were more hesitant; some teachers who had preferred individual work at the beginning of the year, stated that they were uncertain at the end of it. There was agreement about the need to combine individual work with team work, and about the idea that both types of work contribute to students' progress.

Acquiring skills and understanding concepts

Students were asked to define some of the unit's basic concepts and indicate differences between these concepts and similar concepts (abstract data types, a library unit as opposed to a module, implementation as opposed to interface). This examination revealed that in some cases students believed they understood the concept, but their definition showed that this was not so. We may distinguish between two groups of concepts: "Theoretical" concepts ('specification' and 'module') and "practical" concepts implemented in Pascal ('library unit', 'implementation' and 'interface'). The students' understanding of the practical concepts improved immensely, because they learned them "hands on" and used them extensively throughout the year. Students' understanding of all concepts related to efficiency and to the structure of programs improved also. The principle of information hiding, which was totally unfamiliar to 60% of the students at the beginning of the year, was also understood by most of them by the year's end. However, the theoretical concepts were less clear to the students, maybe because they had been used in class less frequently.

Defining abstract data types

An "abstract data type" was defined in the study material as a collection of data of a particular type and a collection of operations defined on it. The students were asked to define the concept and give several examples. Two phenomena were particularly apparent in the students' answers: 1. A partial definition of the concept, defining the abstract data type as a collection of data. This is a very intuitive definition which relies upon knowledge of the Pascal language. Perceiving the overall picture of a collection of data and operations upon them requires a high level of understanding which is not possessed by all students even in advanced stages of their studies. 2. Distinguishing between predefined data types and new data types. Predefined data types, such as integer, real, are abstract data types in the fullest sense, and yet many students tend to consider only new types as abstract data types. This was manifest in both definitions and examples. The term "something new" recurred in many interviews. Almost all examples given by students were of new types rather than types that already existing in the language. This may have been caused by the notion that types existing in the language are very "earthly" and "well grounded", and therefore may not be considered "abstract". All students who answered that an abstract data type is a new data type did not give existing types as examples - showing consistency in their answers.

The interviews suggest that there is no direct correlation between understanding the concept and the student's stage of progress within the unit. Some students gave an accurate definition of the concept at the beginning of the year, while others did not understand it even at the end of their studies. It is important to note that most students were quite capable of handling problems requiring them to define abstract data types - even when they did not fully understand the concept itself. This probably resulted from the acquisition of "technical" skills through exercises and examples, regardless of the correctness of their definition.

Designing algorithms
The interviewed students were asked to design a general algorithm in order to solve the problem presented to them. The algorithm for this specific problem was simple, yet some students found it difficult to design a general top-down solution. These students did not think of a general algorithm that handles various possible inputs and routes them to the appropriate handling procedure. Instead, they focused on small details and immediately began to deal with specific examples. Such a reaction may result also from the pressure of the interview and inexperience in handling the given problems.

Reusing structures
In the interviews we tried to examine the students' ability to define general structures and reuse them properly. The students' answers show a maturity of their approach to problem solving reusing general structures. Many of the students who began by thinking of a specific solution progressed of their own accord to using reusable general structures. The students utilized different structures for solving the problem presented to them.

Conclusions

The unit aims to provide students with capabilities of analyzing problems and abstract thinking skills. Students' questionnaires, exams they took, and interviews, indicate that students acquired these skills. They learned to design solutions for algorithmic problems and to use both new and existing library units for implementing these solutions. Most students were able to use the tools acquired even if they were not able of defining them correctly.

Integrating theoretical and practical aspects was one of the principles that guided the developers. Programming in itself was not the focus of the unit - it served only as a tool for implementing and internalizing the theoretical concepts included in the syllabus. Many teachers and students expressed dissatisfaction with the heavy emphasis on the theoretical framework, and the relative neglect of practicality and usability. Here teachers have a crucial role. A teacher who is well acquainted with the materials, and senses “the spirit of the study unit” passes his knowledge and feelings on to his students. Such a teacher formulates examples and exercises that arouse the students’ interest, and teaches the material in the right manner. An insecure teacher who follows the book to the letter, does not offer the additional value of his or her own experience. The students of such a teacher learn rules and definitions by heart, but do not always know how to apply them in new situations.

An important means for internalizing the theoretical ideas is the case study. When the case study was first introduced into the syllabus, many teachers were somewhat intimidated by its challenge, and thus an artificial barrier was created between the case study and the rest of the study program. Despite difficulties in implementing the case study, it has clearly emerged as an important didactic aid for practicing and internalizing the unit’s theoretical contents. The teachers’ apprehensions also seem to be abating gradually.

One of this unit’s goals was developing the ability to define complex tools from the basic tools provided by the programming language and to reuse them as needed. The study shows that this goal was achieved.

There were also consequences drawn related to the structure of the matriculation examinations. Since this is only of local importance we will not elaborate on it here.

To conclude we would like to add that despite the fact that the research was related to a unit within the framework of a computer science high school program, its conclusions can be adopted to college or university level, while the unit itself can be “upgraded” with only little effort, to college or university level.

Acknowledgment
This paper reflects the efforts of many people too numerous to list. We would especially like to thank Ofra Brandeis and Meir Komar, who devoted a great deal of time and knowledge to the advancement of our research.

References

1. Fleury, A. E. (1993). Students Beliefs about Pascal Programming, *Journal of Educational Computing Research,* 9 (3), 355-371.
2. Gal-Ezer, J., Beeri, C., Harel, D., & Yehudai, A. (1995). A High-School Program in Computer Science, *Computer* 28 (10), 73-80.
3. Linn, M. C., & Clancy, M. J. (1992). The Case for Case Studies of Programming Problems", *Comm. Assoc. Comput. Mach.* 35 (3), 121-132.
4. Piaget, J. (1948, 2nd ed. 1974). *To understand is to invent: The future of education*, NY: Viking.
5. Perkins, D. N. (1992). *Smart schools - From training memories to training minds*, NY: The Free Press.
6. Winogard, T. (1995). From Programming Environments to Environments for Designing, *Comm. Assoc. Comput. Mach.* 38 (6), 65-74.
7. Brandeis, O. et al. (1997). *Software Design* (**in Hebrew**).
8. Brandeis, O. et al. (1997). *Teachers' Guide* (**in Hebrew**).
9. Brandeis, O. et al. (1997). *Case Study* (**in Hebrew**).