The Dawn of Software Engineering Education

Andrii M. Striuk[1][0000-0001-9240-1976] and Serhiy O. Semerikov[1,2,3][0000-0003-0789-0272]

[1] Kryvyi Rih National University, 11, Vitalii Matusevych Str., Kryvyi Rih, 50027, Ukraine
[2] Kryvyi Rih State Pedagogical University, 54, Gagarina Ave., Kryvyi Rih, 50086, Ukraine
[3] Institute of Information Technologies and Learning Tools of NAES of Ukraine,
9, M. Berlynskoho Str., Kyiv, 04060, Ukraine
andrey.n.stryuk@gmail.com, semerikov@gmail.com

Abstract. Designing a mobile-oriented environment for professional and practical training requires determining the stable (fundamental) and mobile (technological) components of its content and determining the appropriate model for specialist training. In order to determine the ratio of fundamental and technological in the content of software engineers' training, a retrospective analysis of the first model of training software engineers developed in the early 1970s was carried out and its compliance with the current state of software engineering development as a field of knowledge and a new the standard of higher education in Ukraine, specialty 121 "Software Engineering". It is determined that the consistency and scalability inherent in the historically first training program are largely consistent with the ideas of evolutionary software design. An analysis of its content also provided an opportunity to identify the links between the training for software engineers and training for computer science, computer engineering, cybersecurity, information systems and technologies. It has been established that the fundamental core of software engineers' training should ensure that students achieve such leading learning outcomes: to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of language, instrumental and computational tools for software engineering; know and apply the appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development; put into practice the software tools for domain analysis, design, testing, visualization, measurement and documentation of software. It is shown that the formation of the relevant competencies of future software engineers must be carried out in the training of all disciplines of professional and practical training.

Keywords: software engineering, professional training, software, specialist training model, standard of higher education.

1 Introduction

On October 29, 2018, by Order of Ministry of Education and Science of Ukraine № 1166, the higher education standard for specialty 121 "Software Engineering" for the first (bachelor's) level of higher education was approved and implemented

(Standard 2018) [88]. The Standard of higher education contains competencies that determine the bachelor's training specifics in specialty 121 "Software Engineering" and program results of training. The introduction of the standard was preceded by a long work of domestic software engineering (SE) specialist community, the result of which was the formation of future SE specialist' general and special competency-based system.

The rapid development of models, methods and tools of SE and the IT industry as a whole raises the question of whether the proposed competency-based system is sufficient to form a modern SE specialist capable of professional development and self-improvement throughout life. A positive answer to this question will contribute to improving Ukraine's position in the IT international labor market, and a negative one will preserve the current state of technology development and the programmed lag of Ukraine's IT education from other states.

To answer this question, it is necessary to determine the ratio of the stable (fundamental) and mobile (technological) component in the content of vocational training of specialists in SE. As one of the founders of SE Brian Randell (born 1936) pointed out that "Those who cannot remember the past are condemned to repeat it. So I hope that you will ... to pay a little more attention to the past, ... to take some time to read or re-read, for example, the original 1968 NATO Report, ... before you resume work on inventing yet another new language or [programming] technique" [7, p. 8]. That is why an analysis of the main stages of development of software engineering as a branch of knowledge was presented, the fundamental components of the training of future software engineers were identified and development trends for this industry for the next decade were determined in our previous work [9].

The purpose of the article is to analyze the first software engineering specialists training model developed under the guidance of Friedrich Ludwig Bauer, and to establish its compliance with the current state of SE development and the new higher education standard in specialty 121 "Software Engineering".

2 The first software engineering training model (1972)

According to the results of the NATO conference on software engineering 1968 [6] and 1969 [2] an international experts group led by Friedrich Ludwig "Fritz" Bauer (1924-2015) developed "Advanced Course on Software Engineering" [1] in late 1971 – early 1972. The Advanced Course took place February 21 – March 3, 1972, organized by the Mathematical Institute of the Technical University of Munich and the Leibniz Computing Center of the Bavarian Academy of Sciences. In the Preface to the materials of this training program for SE Bauer indicates that the book [1] is the first step towards the concentration and systematization of relevant teaching materials: "Our intention in the planning of this course was to cover as much as we can at the moment of all the aspects of the theme [SE], and to contribute further to the systematization of the field. ... we think it is essential to point out where the ideas of software engineering should influence Computer Science and should penetrate in its curricula ... to your students in an academic environment" [1, p. 2].

The developed training program for SE consisted of four sections.

The first section – "Introduction" – contained two lectures. In the first lecture by Keith William "Bill" Morton (born 1930), using concrete examples, describes the problems that led to the software crisis and identifies 8 main directions for overcoming them when developing application software packages that correspond to learning outcomes of the 2018 standard (Table 1).

Table 1. Directions for overcoming a software crisis

Main directions for overcoming the software crisis according to Morton	**Programmatic results for SE professionals according to Standard 2018**
project management – training of staff in appropriate programming techniques; setting up standards; sub-dividing work into manageable parts; monitoring progress and quality	PR22 – to know and be able to apply project management methods and tools
product definition – specifying its function; defining user image; effects of host operating system	PR09 – to know and be able to use methods and means of collecting, formulating and analyzing software and PR11 – to select the initial data for design, guided by formal methods for describing requirements and modeling
design and implementation	PR10 – to conduct a pre-project survey of the subject area, a system analysis of the design object, PR12 – to put into practice effective approaches to software design, PR14 – to put into practice the software tools for domain analysis, design, testing, visualization, measurement and documenting of software and others
application of different levels problem-oriented languages or abstract machines hierarchies, most suitable for solving applied problems	PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of language, instrumental and computational tools for software engineering and PR15 – motivated to choose programming languages and development technologies for solving problems of creating and maintaining software
documentation – selecting levels, methods and automatic aids; controlling quality; disseminating and updating	PR05 – to know and apply the appropriate mathematical concepts, domain, system and object-oriented analysis and mathematical modeling methods for software development, PR14 – to put into practice the software tools for domain analysis, design, testing, visualization, measurement and documenting of software and PR16 – to have the skills of team development, coordination, design and release of all program documentation types
testing – generation of test data; use of test beds	
performance measurement – simulation; measurement tools; monitoring and optimisation	
maintenance and enhancement	

At the end of the lecture, Morton answers the question posed in title – "What the Software Engineer Can Do for the Computer User": improvements to the computer systems that computer user has to use; and tools and techniques that software engineer can make use of in his own work [1, pp. 4–11].

Jack Bonnell Dennis' (born 1931) the second lecture of the first section "The Design and Construction of Software Systems" examined such basic concepts as "computer systems", "software systems", "their hierarchy", "system software", and "application software", methods for describing software systems, their function, correctness, performance and reliability, as well as software projects.

At the beginning of the lecture, the author sets out his vision of SE as the application of principles, skills and art to the design and construction of programs and systems of programs, focusing specifically on the sound principles of SE and the extent of their applicability.

Dennis defined a computer system as a combination of hardware and software components that provides a definite form of service to a group of "users" [1, p. 13]. Therefore, for programming subsystems using the example of the Basic language, the author identifies at least three distinct computer systems and corresponding user groups (consumers of certain services):

1. the computer hardware, whose "users" are operating system implementers;
2. hardware plus operating system, whose "users" are subsystem implementers;
3. hardware, operating system and Basic language subsystem, which services are provided to users of this language.

Based on this unity, the author defined software systems as the software and hardware components that must be added to a specific computer system, called the host system, in order to realize some desired function. Thus, the hardware and the operating system form the base computer system for a software programming system in the Basic language.

The systems hierarchy is formed by expanding them or defining a new way of interacting (linguistic level) with them through translation or interpretation of commands.

System software – a collection of system programs usually forms a hierarchy of software systems having these properties:

1. The collection of programs is implemented under one authority.
2. The hierarchy of software systems defines a single linguistic level which applies to all users of the collection of programs.
3. Inner linguistic levels of the hierarchy are hidden from the user.
4. The outer linguistic level of the hierarchy is "complete" for the goals of the implementing authority.
5. The primary means of defining new linguistic levels is partial interpretation.

Application software – an application program or software system usually has these properties:

1. The programs are expressed in terms of a "complete" linguistic level.
2. The programs define a new linguistic level by extension, translation, interpretation, or by some combination of these techniques.
3. The linguistic level defined by the program or software system is inadequate for defining further linguistic levels.

4. A variety of such programs or software systems are available to clients of an installation, and are often implemented under different authorities.

J. B. Dennis determines the description of the software system through the description of its software component (expressed in a well-defined programming language), hardware component (using a model that reflects the behavior of the hardware component in all normal operating conditions of the software system), the base system and the linguistic level at which a software system should be implemented.

The goals of a software system designing are expressed in its desired properties: functionality (as the correspondence desired of output with input), correctness (using a structural programming style that makes program correctness self-evident, or using proof of correctness of software systems or components), performance (the effectiveness with which resources of the host system are utilized toward meeting the objective of the software system) and reliability (the ability of a software system to perform its function correctly in spite of failures of computer system components).

Dennis' lecture concludes with a description of the typical tasks that arise when developing large software projects.

2018 Standard defines the following programmatic outcomes relevant to this lecture:

PR03 – to know the main processes, phases and iterations of the software life cycle;

PR05 – to know and apply the appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development;

PR06 – the ability to choose and use the appropriate methodology for creating software;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering;

PR10 – to conduct a pre-design survey of the subject area, a system analysis of the design object;

PR11 – to choose the source data for design, guided by formal methods for describing requirements and modeling;

PR12 – to put into practice efficient approaches to software design;

PR13 – to know and apply methods for developing algorithms, designing software for data structures and knowledge;

PR14 – to apply in practice the software tools for domain analysis, design, testing, visualization, measurement and documenting of software;

PR15 – motivated to choose programming languages and development technologies for solving problems of creating and maintaining software;

PR22 – to know and be able to apply methods and means of project management.

The second section – "Descriptional Tools" – contained five lectures. In the first lecture "Hierarchies" Gerhard Goos (born 1937) discusses the methodological foundations of the decomposition of software systems and their application to the development of software and programming languages.

Goos suggests starting the software systems designing with a description of the problem to be solved and the available host system (as interpreted by J. B. Dennis). The task can be formally represented by an abstract machine, which from the input data using the program for this machine produces some source data that solve a certain part of the problem [1, p. 29]. At the first stage (general system design), a set of software components is determined by specifying their external interfaces, each of which solves a part of the original problem, realizing it using some functions of an abstract machine. The internal behavior of each component is determined at the second stage (detailed system design).

The results of general design can be represented as a network of components – a directed graph of arbitrary complexity. The presence of fatal cycles in this reflects the complexity of the design process. Goos calls the principle of structuring systems in the form of a partial ordered set of layers (including a tree-like or linear structure) hierarchical beautification and indicates that its application allows you to divide the system into components in such a way that a picture of their interconnections is obtained and the contribution of each component to solving the general problem of the system is clearly determined. Each layer can be implemented by a specific abstract machine, the sequence of which forms a system, and each next (lower) layer provides all the necessary tools for the implementation of the previous (higher) layer, and the last layer implements an abstract machine, which is identical to the base system. Programming language is defined for each level of abstraction that provides access to all the functions of an abstract machine of a given level.

Goos considers hierarchical design as a means of engineering software development and production. The main assumption of the author is that operations and data structures which are present at the A_{i+1} level, but are absent at the A_i level, cannot be used by programs running at the A_i level. This assumption provides the possibility of using hierarchical ordering as well as testing and debugging tools [1, p. 44-45].

Corresponding lecture program results of training according to 2018 Standard:

PR05 – to know and apply appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering;

PR17 – to be able to apply the methods of component software development.

From an engineering point of view Goos examined how programming languages affect the program development process and its properties in the second lecture "Language Characteristics: Programming Languages as a Tool in Writing System Software".

Just as natural language affects the thinking of the person who uses it, programming language influences its user at least with respect to the following:

- The conceptual understanding how a problem can be solved by computer
- The range of problems which can be attacked by programming
- The set of basic notions available for programming
- The style of programming (clarity, robustness, readability etc.)
- The meaning of "portability"
- The meaning of "efficiency"

The lecture established a correspondence between the properties of the languages and the properties of the programs described by them, and defined some characteristics of "good" programming languages, the main of which is to encourage the programmer to write correctly (from an engineering point of view) the designed programs. The author paid special attention to the characteristics of high-level system programming languages. Goos primarily refers the means of improving the quality of system software engineering (creating "good" programs) to structural programming tools (in particular, modularity), hierarchical ordering, the use of nesting and visibility, as well as parallel execution.

Corresponding lecture program results of training according to 2018 Standard:

PR06 – the ability to choose and use the appropriate methodology for creating software;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering;

PR15 – motivated to choose programming languages and development technologies for solving problems of creating and maintaining software;

The third lecture of the second section "Low Level Languages: Summary of a Discussion Session" is a synthesis of round table materials prepared by M. Griffiths (France), in which F. L. Bauer, G. Goos, M. Griffiths, and other SE experts participated. Answering the question of why creating a new low-level programming language, the panelists indicate that existing machine languages (assemblers) do not provide support for efficient memory allocation and compliance with the programming style, which aims to increase the reliability of software. The discussion participants set the following requirements for new low-level programming languages: the availability of flow control tools (cyclic and conditional statements), support for modularity (at least at the procedure level with parameter passing), data structures (with the possibility of type conversion, indexing, address arithmetic). Moreover, such languages can be either machine-oriented or problem-oriented (such as Forth).

It was proposed to measure the effectiveness of low-level programming languages by three indicators: the complexity of the programmer's labor, computer time, and memory capacity. The panelists noted regarding the programming style the vital role of training future system programmers using appropriate tools and a good programming style. At the time of the course, the discussion participants considered a successful example of a low-level language that meets these requirements, namely a language that was used to develop the MULTICS – PL / I operating system, more precisely, its early EPL dialect, which significantly influenced the design of the programming language, and today it is a reference implementation of the concept developed during the discussion – C.

Corresponding lecture program results of training according to 2018 Standard:

PR06 – the ability to choose and use the appropriate methodology for creating software;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering;

PR15 – motivated to choose programming languages and development technologies for solving problems of creating and maintaining software;

The fourth lecture “Relationship Between Definition and Implementation of a Language” is related to the main area of M. Griffiths research – the development of compilers: “The non-specialist sometimes accuses both computer scientists and software engineers of spending all their time on discussions about languages, to the detriment of all the ‘real’ problems. Whilst this accusation is net without foundation, it must be clearly understood that language is central to the whole problem of software engineering. If we cannot supply powerful, well-defined, understandable languages with corresponding, economic implementations on existing computing equipment, then the programmer can hardly be expected to express himself in a way which will permit us to use the term which is the theme of this school.” [1, p. 77]. The lecture discussed the impact of language definition on the programs written in the language and on the methods used to execute these programs in the computer.

The author identifies three points of view on the definition of language: 1) the user point of view, which focuses on the task, and not on the features of the language description (language description as a guide); 2) the compiler developer point of view, which seeks to implement all the language described features, even those that very little will be used (language description as scripture); 3) the researcher point of view, for whom the language description is a complete, formal, mathematically correct set of statements that reflect traditional mathematical principles of minimizing the interconnected axioms number.

According to the author, the main thing in determining the language is the absence of side effects. He distinguishes two components of semantics for this purpose: static semantics is a part of a programming language semantics that is independent of program execution and determined at compile time (such as the relationship between the use of an identifier and its announcement); dynamic semantics takes into account real object manipulations and their values when program execution.

The leading is the grammatical approach in determining the language syntax, particularly using LL (1)-grammars to implement context-free languages. M. Griffiths gave a range of examples which demonstrates the influence of the method of determining the language on its implementation, showing that the ease, safety, and efficiency of the language implementation directly depends on its definition: “It is this idea which prompts us to suggest that the definition of a language should foresee its implementation, and that the implementation should be strongly directed from the very start. There is here a parallel with the architecture of computer hardware, which is likely improve conceptually each time the hardware engineer works close enough to the software engineer and hence considers the use to which the machine may be put. The use to which a language definition is put is in the first instance the implementation, We consider, therefore, that a language definition should be in terms of an idealised implementation, or at least be accompanied by an “implementers’ guide” [1, pp. 99–100].

The following methods for determining the language are discussed in the lecture:

- two-level grammar (a formal grammar that is used to generate another formal grammar), such as the Van Wijngaarden grammar for Algol 68 in conjunction with a stylized natural language description of a program interpretation on a hypothetical computer;
- Vienna Definitions (Vienna Definition Language – VDL), today, it is better known as the Vienna Development Method (VDM) – a set of technologies for modeling systems, analyzing created models and moving to detailed design and programming), which used operational (connotative) semantics that is a way of describing a language using sequences of calculation steps, and it is very closely connected with the implementation of the system in a programming language, since the calculation steps are described in the language of some computer;
- expanding languages that consisting of a relatively simple but self-sufficient base language, which itself contains extension mechanisms that make it possible to define new operators or data types.

Corresponding lecture program results of training according to 2018 Standard:

PR05 – to know and apply appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering;

PR12 – to put into practice efficient approaches to software design;

PR13 – to know and apply methods for developing algorithms, designing software for data structures and knowledge.

The last lecture in the second section “Concurrency in software systems”, written by J. B. Dennis, focuses on large modular programs such as operating systems, compilers, or real-time control programs, many interacting parts or modules. Due to the size of these, programs, it is essential that the parts be represented in such a way that the descriptions of the parts are independent of the pattern in which they are interconnected to form the whole system, and so the behavior of each part is unambiguous and correctly understood regardless of the situation in which it is used. For this to be possible, all interactions between system parts must be through explicit points of communication established by the designer of each part. If two parts of a system are independently designed, then the timing of events within one part can only be constrained with respect to events in the other part as a result of interaction between the two parts. So long as no interaction takes place, events in two parts of a system may proceed concurrently and with no definite time relationship among them. Imposing a time relation on independent actions of separate parts of a system is a common source of overspecification. The result is a system that is more difficult to comprehend, troublesome to alter, and incorporates unnecessary delays that may reduce performance. This reasoning shows that the notions of concurrency and asynchronous operation are fundamental aspects of software systems. According to Dennis, such considerations show that the concept of parallelism and asynchronous operation are fundamental aspects of software systems.

The lecture discusses system models presented as sets of simultaneously acting subsystems interacting with each other through certain communication mechanisms. The author shows that if the interaction between the subsystems is subject to certain environmental conditions, the determinism of the subsystems guarantees the determinism of the entire system (the ability to give the same results for different launches with the same input data). Such system description is performed using Petri nets: the lecture ends with an example of their application to systems with parallel processes that interact by means of semaphores, using the synchronization primitives P and V by Edsger Wybe Dijkstra.

Corresponding lecture program results of training according to 2018 Standard:

PR05 – to know and apply appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering.

The third section "Techniques" contained four lectures. In the first lecture "Modularity" J. B. Dennis deepens the discussion of modularity concept that begun in the previous lecture, which is closely related to structural programming as a property of computer systems: "A computer system has modularity if the linguistic level defined by the computer system meets these conditions: Associated with the linguistic level is a class of objects that are the units of program representation. These objects are program modules. The linguistic level must provide a means of combining program modules into larger program modules without requiring changes to any of the component modules. Further, the meaning of a program module must be independent of the context in which it is used" [1, p. 130].

Dennis considers modularity both in the context of the previous lecture (competition with communication) and in the procedural context, when one module calls another with a certain input data set, eventually obtaining the result. The lecture considers issues arising in the construction of software modules that require the ability to create, expand and modify structured data. It is concluded that in order to achieve modularity, a computer system must determine a linguistic level that provides an appropriate basic representation for structured data: 1) any data structure may occur as a component of another data structure; 2) any data structure may be passed (by reference) to or from a program module as an actual parameter. 3) a program module may build data structures of arbitrary complexity. At this level, the memory should be addressed not by elements of a predetermined length, but by data structures. The author considers the main memory size of the computer system to be the main limitation on the module size, and virtualization is the way to overcome it.

Significant support for modularity can come from a correctly designed operating system – for example, the author gives unique characteristics of Multics that contribute to the implementation of the concept under discussion:

1. A large virtual address space (approximately 2^{30} elements) is provided for each user.

2. All user information is accessed through his virtual address space. No separate access mechanism is provided for particular sorts of data such as files.
3. Any procedure activation can acquire an amount of working space limited only by the number of free segments in the user's address space.
4. Any procedure may be shared by many processes without the need of making copies.
5. Every procedure written in standard Multics user languages (FORTRAN, PL/I and others) may be activated multiply through recursion or concurrency.
6. A common target representation is used by the compilers of two major source languages – PL/I and FORTRAN.

These achievements are Multics major contributions toward simplifying the design and implementation of large software systems. They were made possible by building the Multics software on a machine expressly organized for the realization of a large virtual memory and shared access to data and procedure segments [1, p. 161].

The last part of the lecture informally presents semantic concepts of the linguistic level (common basic language), which can serve for the general presentation of program modules expressed by various programming languages.

Corresponding lecture program results of training according to 2018 Standard:

PR06 – the ability to choose and use the appropriate methodology for creating software;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering;

PR15 – motivated to choose programming languages and development technologies for solving problems of creating and maintaining software;

PR17 – to be able to apply the methods of component software development.

The second lecture "Portability and Adaptabilty" by Peter Cyril Poole (1931 – 2017) and William M. Waite (circa 1936) begins with the definition of portability as measure of the ease with which a program can be transferred from one environment to another, and adaptability as measure of the ease with which a program can be altered to fit differing user images and system constraints. The major distinction between the two concepts is that adaptability is concerned with changes in the structure of the algorithm, whereas portability is concerned with changes in the environment.

The main way to increase program mobility is to use high-level languages, provided that:

- the basic operations and data types required by the problem are available in the chosen language;
- the chosen language has a standard definition, and this standard definition is widely implemented;
- care is taken to avoid constructions which are accepted in the local dialect, but prohibited by the standard [1, p. 187].

Another way is to use abstract machines on which the mobile program will run. In this case, compilation from a high-level language is first performed on the abstract machine language (such as UNCOL – Universal Computer Oriented Language), which must necessarily support at least integers and integer arithmetic, comparison and relational operations, symbolic input and output, as well as real numbers and real arithmetic, strings (connection, substring selection, lexicographic comparison), input and output of memory images, labels and means of control transfer, declarations, arrays and records, conditional and cyclic operators, procedures and blocks. At the end of the lecture, the authors built abstract machines hierarchy that provide mobility and adaptability and level out some of the disadvantages of UNCOL.

Corresponding lecture program results of training according to 2018 Standard:

PR05 – to know and apply appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering;

PR13 – to know and apply methods for developing algorithms, designing software for data structures and knowledge;

PR15 – motivated to choose programming languages and development technologies for solving problems of creating and maintaining software.

At the beginning of the third lecture “Debugging and Testing”, P. C. Poole points out that proof of program correctness by formal methods (such as VDM) is too long to find errors, so a good software engineer should plan the testing and debugging phase [1, p. 279] (moreover, testing in no case does not show the absence of errors – only their presence). The standards of software documentation which are discussed in a separate lecture, and the “readability” of program code is played an important role in this, in particular, the presence of meaningful comments is useful for debugging and indispensable for maintaining and improving software. The program code must be accompanied by benchmarks and data that must be changed with the code.

Poole introduced the concept of “guard code” is the use of logical expressions to limit computation options [1, p. 287], which he suggests actively using in conjunction with the “principle of exceptional suspicion”, which consists in the fact that the module should not use any data transmitted to it through the interface without first checking its compliance with the interface specification.

The main methods for fixing at that time were analysis of posthumous dumps, debugging output, direct and reverse tracing of the program code part, compilation with the inclusion of data for debugging, code documentation (reformatting the text in order to highlight the structural elements of the code, creating tables containing all procedures, labels and announcements, creating an index of procedure calls, following labels and variable references, visualizing control flows, etc.). The author considers the interactive setup managed by the user as a promising tool (online debugging).

Corresponding lecture program results of training according to 2018 Standard:

PR14 – to apply in practice the software tools for domain analysis, design, testing, visualization, measurement and documenting of software;

PR19 – to know and be able to apply software verification and validation methods;

PR20 – to know the approaches for evaluating and ensuring the quality of software.

Dionysios C. Tsichritzis (born 1943) considers the design and construction of reliable software in the last lecture of the section “Reliability”. The author begins the review with the remarks of E. V. Dijkstra “Testing can show the presence of errors and not their absence” and B. Randell “Reliability is not an add-on feature”, emphasizing the main idea of SE is to provide means that will enable the individual medium ability to create quality software [1, pp. 319-320].

Tsichritzis postulates a number of provisions that affect the reliability of software development:

1. different programming languages prompt programmers to various types of errors, which he calls characteristical;
2. the compiler must identify semantic inconsistencies in the programs text;
3. programming style in particular:

   - using appropriate variable names and structuring program code;
   - the use of methods such as cross-checking, checking ranges of variable values, checking the consistency of data changes, observing the uniqueness of names, phased data processing, including error codes in the results;

4. the impact of protection (a managed software environment with clearly defined rules and restrictions) by diagnosing errors as a violation of protection and isolating errors in the modules in which they took place;
5. proof of the program correctness by informal (by introducing checks on the values of input, intermediate, and output variables) and by formal (for example, by calculating first-order predicates) methods;
6. program design aimed at improving reliability:

   - structuring a program to facilitate testing;
   - informal verification of the some program parts logical correctness;
   - application of synchronization and process interaction tools;
   - using a hierarchy of abstract machines;

7. ensuring reliability during program operation by:

   - data integrity using partial or full memory dumps;
   - duplication of key data in different repositories;
   - accounting for hardware failures;
   - availability of program resumption to minimize the failures impact.

Discussing ways to protect the system as a whole, processes and user data, Tsichritzis introduces the concepts of domain (active and potentially aggressive entity) and object (passive and potentially vulnerable entity), giving examples of their implementation through processes and files. At the virtual machine abstraction level, reliability violations are a violation by the user of the operation of their own virtual machine or virtual machines of other users. The author uses a matrix apparatus for allocating resources between parallel processes in conditions of a lack of resources to describe this process. The author offers randomly generated names of synchronization objects (magnum – magic numbers) among the methods used to this day. The lecture discusses in detail the data protection mechanisms in file systems that should be implemented in the kernel of the operating system.

The author interprets the concept of data security (information security, data security or cybersecurity) as controlling access to privileged data stored in large scalable data banks in three categories [1, p. 357]:

1. information privacy involves issues of law, ethics and judgment controlling the access of information by individuals;
2. information confidentiality involves rules of access to data;
3. information security involves the means and mechanisms to ensure that privacy decisions are enforceable.

Tsichritzis distinguishes between protection and data security in the following way: protection concerns only access control to data in the operating system without taking into account the nature of data, and for information security tasks the information system is considered as a whole, and not just a computer-based system. The active elements of the information system are people, and the nature and content of the data are taken into account when providing access to them. That is why the author considers examples of information security tasks not on file systems, but on database management systems.

The author considers the following important tasks of information security:

a. at the login level: user identification, authentication, monitoring of resource use;
b. at the file system level: determining available files, user identification, password-controlled file properties, cryptographic protection;
c. at the data protection level: data separation of various users, data integrity, backup, reliable deletion of protected data;
d. restrictions on use (in particular, forced logout due to non-standard actions).

Corresponding lecture program results of training according to 2018 Standard:

PR05 – to know and apply appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development;

PR06 – the ability to choose and use the appropriate methodology for creating software;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering;

PR08 – to be able to develop a human-machine interface;

PR11 – to choose the source data for design, guided by formal methods for describing requirements and modeling;

PR14 – to apply in practice the software tools for domain analysis, design, testing, visualization, measurement and documenting of software;

PR18 – to know and be able to apply information technology for processing, storage and data transfer;

PR19 – to know and be able to apply software verification and validation methods;
PR20 – to know the approaches for evaluating and ensuring the quality of software;
PR21 – to know, analyze, choose, and skillfully apply information security (including cybersecurity) means and data integrity in accordance with the tasks being solved and the software systems being created.

The fourth section "Practical aspects" is the largest in the entire course and includes six lectures. D. Tsichritzis formulates the general goal of software project management in the first lecture of "Project management" that is "production of the desired product for specific design goals, specifications and available resources" [1, p. 375]. To achieve this goal the author suggests setting up developer communications (through the use of common documentation, frequent informal communication and regular meetings of small development groups), organizing developers (by identifying a person who has a systematic view of the project, attracting developers to other components of the project and reducing management formalism) and set control points.

Among the software project development tools Tsichritzis pays special attention to information system modeling tools (for example, Petri nets are such a tool for describing parallel processes) and computer-aided design software systems, some of which (such as Project LOGOS) are significantly ahead of their time. The main difference between small and large software projects, in his opinion, is the number of management levels: there should be at least two of them. The issues of reducing the time for implementing software projects and requirements for managers of large projects are discussed at the end of the lecture.

Corresponding lecture program results of training according to 2018 Standard:
PR03 – to know the main processes, phases and iterations of the software life cycle;
PR04 – to know and apply professional standards and other normative legal documents in the field of software engineering;
PR05 – to know and apply appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development;
PR11 – to choose the source data for design, guided by formal methods for describing requirements and modeling;
PR12 – to put into practice efficient approaches to software design;
PR14 – to apply in practice the software tools for domain analysis, design, testing, visualization, measurement and documenting of software;
PR16 – to have the skills of team development, coordination, design and release of all program documentation types;
PR18 – to know and be able to apply information technology for processing, storage and data transfer;
PR22 – to know and be able to apply methods and means of project management;
PR23 – to be able to document and present the results of software development.

In the second lecture of the section "Documentation", G. Goos poses 8 questions that various types of documentation should answer (Table 2) [1, p. 386].

Table 2. Types of documentation (by G. Goos)

Questions	**user's guide**	**conceptual description**	**design documentation**	**product documentation**
How to use a program?	▨			
What is the state of the project?		▨	▨	▨
What are the overall specifications of the project?		▨	▨	
Which models are used to subdivide the program and to interface different modules?		▨	▨	
Which basic models are used for the modules?		▨	▨	
Flow of control and flow of data through the program			▨	
Detailed description of data.				
What is the meaning of the error-messages?	▨			

Short but comprehensive answers to these questions are one of the conditions for the success of the project – "if programmers cannot get a clear understanding how their work is related to the work of others they must fail because they usually start from wrong assumptions about their environment" [1, p. 387].

Creating a user manual (introductory manual, reference manual and operator's guide) starts first and ends last. The Introductory manual not only describes the standard ways of using the program, serving as a kind of "program cookbook", but also is the basis for its advertising and sale.

The conceptual description is developed during the project as its general description, the project documentation describes the current state of the project at the design phase and is the basis for the design phase, and the product documentation (including with the program text) describes the current state of the project at the design and maintenance phases. Inclusion in the final program text is necessary for cross-referencing interface modules, data, and algorithms.

Keeping documentation up-to-date is an important task for which Goos suggests using distributed text-based documentation systems with timestamp support.

Corresponding lecture program results of training according to 2018 Standard:

PR04 – to know and apply professional standards and other normative legal documents in the field of software engineering;

PR14 – to apply in practice the software tools for domain analysis, design, testing, visualization, measurement and documenting of software;

PR16 – to have the skills of team development, coordination, design and release of all program documentation types;

PR23 – to be able to document and present the results of software development;

PR24 – to be able to calculate the cost-effectiveness of software systems.

The third lecture "Performance Prediction" is one of the largest in the section. Its author Robert M. Graham (born 1929) was one of the first experts in the field of cybersecurity. He was responsible for security mechanisms, dynamic linking, and other key components of the operating system kernel in the Multics project.

Using the operating systems design and development as an example, the author offers criteria and methods for evaluating the performance of software systems, and also analyzes the impact of these scopes on software development and implementation processes. In this context, the author considers performance as the efficiency of using system resources when realizing software goals. The author emphasizes the importance of modeling systems to assess their efficiency and productivity. According to the author, the model of a software system reflects the relationship between the variables of this system to varying degrees. The model complexity is directly related to the system complexity and the ways to use it. For simple systems, the model will be so simple that it can exist only in the designer imagination. Due to increase of complexity, there is a need to reflect the system model in some exact and formal way. Graham notes that a complete, detailed description of a system, for example, its program code, is also actually the model. As a rule such a model is not useful, because it contains a large amount of redundant information and does not clearly demonstrate the relationship between the main variables: “A model is an abstraction containing only the significant variables and relations. Hence, it is usually much simpler than the system which it models. How much simpler will depend heavily on the expected use of the model and the precision desired in the results of its use” [1, p. 404].

Conceptually, a model is a function or a set of functions in which the system parameters used to characterize system performance are expressed as functions of the main variables of the system. The author emphasizes that “the performance of a system is not a constant number, rather it is a function, or several functions, whose values depend on the input. In order to characterize the performance of a given system we have to express these functions and this expression is a model of the system” [1, p. 404].

Thus, an important component of the professional training of specialists in software engineering is the mastery of the methods and means of formal description of software system models.

The author identifies two basic types of models: analytical and logical. The analytical model does not reflect the system structure. Actually, it is a set of mathematical equations which express the relations which exist between the basic system variables and the performance parameter. These equations are then solved for the dependent variables, that is, for the performance parameters. After solving these equations the system’s performance is completely known since graphs of the performance parameters can be plotted from the resulting mathematical expressions. On other hand a logical model mirrors closely the structure of the system being modeled. It is important to obtain accurate expressions for evaluating performance from this model. However, a logical model often includes mathematical equations which express some of the relations between variables.

Graham provides a description of processor time planning algorithms in operating systems as an example of the analytical model implementation. The author emphasizes the stochastic nature of such a model, but anticipating certain ranges of variables values in the system and the probability of these values, we can evaluate the expected performance of the system.

According to the author, one of the simplest and most obvious logical system model is its representation in the form of a directed graph. Examples of such graphs are flowcharts. In the general case, such a graph nodes are certain system states (variables values), and arcs are the processes or actions that are necessary for the transition from one state to another. We get the opportunity to evaluate and measure system performance by evaluating the time or other resources needed for such transitions.

Graham pays particular attention to simulation models as the most general and flexible way to evaluate the performance of software systems and demonstrates the use of these models in the operating systems design.

Among the problems that arise in the development of performance models, Graham underlines the model adequacy, the characterization of the tasks or desired system properties, and the simulation results interpretation. The author suggests using specialized modeling languages to describe the models, which should contain the ability to describe classes and their attributes, activities and events, as well as support queues, various probability distributions, and tools for collecting and analyzing data. The fundamental nature of this approach is emphasized by the fact that with the 4 general-purpose modeling languages that the author mentions – GPSS, SIMSCRIPT, SIMULA and CSL – only CSL is currently obsolete. Simula 67 became the basis for the development of C ++ and Java, and the latest versions of aGPSS (1.30) and SIMSCRIPT III (Release 5.0) which is generally date from 2019.

Special-purpose modeling languages are modeling languages of operating systems, both existing and those that have not yet been created. Simulation systems for these languages contain information about the hardware and language constructs for its description. The third lecture of the section ends with examples of such languages use.

Corresponding lecture program results of training according to 2018 Standard:

PR05 – to know and apply appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development;

PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering;

PR10 – to conduct a pre-design survey of the subject area, a system analysis of the design object;

PR14 – to apply in practice the software tools for domain analysis, design, testing, visualization, measurement and documenting of software;

PR20 – to know the approaches for evaluating and ensuring the quality of software.

The fourth lecture "Performance measurements" by Calvin Carl "Kelly" Gotlieb (1921 – 2016) has been called the "Father of Computing" in Canada – in 1948 he was part of the first team in Canada to build computers and to provide computing services, and in 1950, he created the first university course on computing in Canada. Gradually, his interests shifted towards the socio-economic consequences of ICT use, which led to his authorship and the chapter on the economics of software engineering.

C. C. Gotlieb indicates that performance measurements are needed when:

- installing a new computing system;

- changing the configuration or "tuning' it to improve throughput;
- comparing systems to determine technological improvements, economies of scale and cost/benefit ratios.

The author refers to measuring performance methods:

1. Establishment of quality indicators (figures of merit) based on the rating (weight) of system components, in particular – empirical Grossch's law in 1953, formulated by Herbert Reuben John Grossch (1918 – 2010): "giving added economy only as the square root of the increase in speed-that is, to do a calculation ten times as cheaply you must do it one hundred times as fast" [3, p. 310]. Gotlieb points out that value should be the common indicator of productivity, formulating Grosch's law as

$$C = K\sqrt{E},$$

   where C is the cost, K is a constant, E is the effectiveness measured in speed, throughput etc. Thus, productivity is proportional to the square of the cost.

   The author proposes for a more accurate assessment to determine the system performance by linking attributes number with each characteristic of the system, taking into account the weight of each attribute determined by the expert assessment method. The overall performance metric is calculated as a weighted sum of characteristics.

2. Launching a set of "kernel", "benchmark" or synthetic tasks. By "core" (kernel) the author understood a general-purpose benchmark program, for each component of which the necessary measurements (runtime, etc.) were made. Then the performance of two computer systems was compared one relative to the other by launching a "kernel" on each of them. A "benchmark" program is a special program designed to evaluate performance (performance test). Performance measurement was a side effect in the "kernel", but it is the main purpose for a "benchmark" program. Synthetic programs are designed to comprehensively check the stability of the system in normal and forced modes. Today, all these types of tasks are considered as a component of performance test (benchmarking).
3. Observing and measuring using hardware and software monitors at three levels:

   - the number of program calls, the estimated time to complete the task, the time spent on for completing of the task stages, the selected parameters of the execution time, kernel utilization, reading and writing, printing, rotation time, selected priorities, cost, diagnostics called at the system level is measured at the user tasks level;
   - the resource distribution, the channels and input/output operations activity, the task queues and system queues length, service time, etc. are measured at the system level;
   - the traffic and task flows, service utilization, resource allocation, operator actions and interventions, user requests, requests and complaints, expense and income statistics are measured at the hardware level.

4. Analytical or simulation modeling of the systems.

The last method is perhaps the only tools available at the software design stage when a preliminary performance assessment is performed. The first three methods are more often used to evaluate existing systems and alternative configurations.

Corresponding lecture program results of training according to 2018 Standard:

PR05 – to know and apply appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development;

PR19 – to know and be able to apply software verification and validation methods;

PR20 – to know the approaches for evaluating and ensuring the quality of software.

In the fifth lecture of “Pricing mechanisms” section by C. C. Gotlieb shows that pricing serves an important role in allocating service resources and rationalizing planning. Price levels are determined by costs, but also by policy considerations. The different methods of setting levels, along with some of the resulting implications and requirements are examined in the lecture.

The author refers to the main cost components as salaries (specialists in management, operational, applications, development) and fringe benefits (pension, insurance, health plan contributions etc.), equipment (purchase or rental payments, maintenance, communication costs, office equipment), supplies (cards, paper, tapes, documentation), software (purchased, leased, developed in-house), site (space, preparation costs, utilities), overhead (use of purchasing and maintenance services, library), miscellaneous (travel, advertising, user manuals, etc.).

The author analyzes the computer services market, which was formed at that time, and offers examples of various pricing models.

Corresponding lecture program results of training according to 2018 Standard:

PR18 – to know and be able to apply information technology for processing, storage and data transfer;

PR24 – to be able to calculate the cost-effectiveness of software systems.

Hans Jørgen Helms (1931–2010) was one of the co-chairs of the 1968 NATO conference program committee and the author of the last lecture in the section “Evaluation in the Computing Center Environment”, which completes the course. In 1965-1974 he was one of the leading employees of the first Scandinavian supercomputer center – Northern Europe University Computing Center at Technical University of Denmark. Working since 1974 as part of the European Commission, he headed the Joint Research Center of the Science, Research and Development Directorate, which he left in 1995 as Director-General.

Helms points out that this lecture relates not so much to software engineering as to the use of designed and developed programs to provide various services to various user groups. Therefore, the author refers to the computer center environment as a people community using the services of a particular computer system: an airline ticket agent uses a seat reservation system; typist uses a text editing system; bank teller uses an online accounting system; a manager uses a management information system; consulting engineer uses standard engineering programs from the terminal in his office; chemist develops programs to solve own research problems; a student solves tasks from a computer science course; programmer develops programs for the customer, etc. [1, p. 504].

Computing Centre environment is a large, distributed user community that accesses the computing services of remote computers through network terminals. Based on the analysis, the author indicates the directions of the corresponding software optimization for various indicators (runtime, service time, etc.) in order to meet the needs of a growing number of users and avoid potential service "bottle-necks".

The author notes the commonality of university computer centers that provide services without considering their profitability with other providers of utilities, such as postal or transport. As was shown in [5], it was precisely such utility computing services that later transformed into cloud technologies [4].

Corresponding lecture program results of training according to 2018 Standard:

PR08 – to be able to develop a human-machine interface;

PR18 – to know and be able to apply information technology for processing, storage and data transfer;

PR21 – to know, analyze, choose, and skillfully apply information security (including cybersecurity) means and data integrity in accordance with the tasks being solved and the software systems being created.

3 Conclusions

1. The first model of training in software engineering developed under the leadership of F. L. Bauer not only summarized existing experience of the early 1970s of training various software engineering components but also systematized it into the relevant training components. Despite the almost fifty-year age of the course materials, a significant part of them became the fundamental foundation of software engineering, determined the further development of the corresponding theory and empirical generalization.
2. The systematic nature (all sections and chapters are interconnected and cross-linked) and scalability (from a two-week intensive to preparing a bachelor's degree) of the proposed training program largely corresponds to the idea of designing evolutionary software. The analysis of the course content not only showed a number of problems regarding the relationship between the stable (fundamental) and rapidly changing (technological) components of the training content, but also provided an opportunity to identify the links between the training of a specialist in software engineering and training in computer science, computer engineering, cybersecurity, information systems and technology.
3. The analysis of the ratio of the program results of training SE specialists according to F. L. Bauer model (1972) and 2018 Standard provided the opportunity to establish that:
   a. the ability to analyze, purposefully search and select information and reference resources and knowledge necessary for solving professional tasks, taking into account modern achievements of science and technology, should be formed in academic disciplines preceding professional practical disciplines of SE specialist

training and developed in the process of professional training and further professional activity;
   b. the acquisition of knowledge of professional ethics codes, understanding of the social significance and cultural aspects of software engineering and their observance in professional activities require separate focused work of the teacher and students both in a separate academic discipline and in the process of professional training;
   c. the fundamental core of SE specialist training should ensure that students achieve these leading learning outcomes: PR07 – to know and put into practice the fundamental concepts, paradigms and basic principles of the functioning of the linguistic, instrumental and computational tools of software engineering; PR05 – to know and apply appropriate mathematical concepts, domain methods, system and object-oriented analysis and mathematical modeling for software development; PR14 – to apply in practice the software tools for domain analysis, design, testing, visualization, measurement and documenting of software. The development of the relevant competencies of future SE specialists must be carried out in teaching all the disciplines of professional training;
   d. F. L. Bauer model (1972) paid insufficient attention to the engineering and socio-economic foundations of the SE specialist professional activities, which are largely reflected in 2018 Standard.
4. Prospects for further research include an analysis of the relationship between the content of the general professional competencies of the bachelor of SE according to 2018 Standard and alternative domestic and foreign standards.

References

1. Bauer, F.L. (ed.). Software Engineering: An Advanced Course. Lecture Notes in Computer Science, vol. 30) (Formerly published 1973 as Lecture Notes in Economics and Mathematical Systems, vol. 81) Springer-Verlag, Berlin, Heidelberg, New York (1975). doi:10.1007/3-540-07168-7
2. Buxton, J.N., Randell, B. (eds.): Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969. Scientific Affairs Division, NATO, Brussels (1970)
3. Grosch, H.R.J.: High Speed Arithmetic: The Digital Computer as a Research Tool. Journal of the Optical Society of America **43**(4), 306–310 (1953). doi: 10.1364/JOSA.43.000306
4. Markova, O.M., Semerikov, S.O., Striuk, A.M., Shalatska, H.M., Nechypurenko, P.P., Tron, V.V.: Implementation of cloud service models in training of future information technology specialists. In: Kiv, A.E., Soloviev, V.N. (eds.) Proceedings of the 6th Workshop on Cloud Technologies in Education (CTE 2018), Kryvyi Rih, Ukraine, December 21, 2018. CEUR Workshop Proceedings **2433**, 499–515. http://ceur-ws.org/Vol-2433/paper34.pdf (2019). Accessed 10 Sep 2019
5. Markova, O.M., Semerikov, S.O., Striuk, A.M.: The cloud technologies of learning: origin. Information Technologies and Learning Tools **46**(2), 29–44 (2015). doi:10.33407/itlt.v46i2.1234

6. Naur, P., Randell, B. (eds.): Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. Scientific Affairs Division, NATO, Brussels (1968)
7. Randell, B.: Fifty Years of Software Engineering - or - The View from Garmisch. arXiv:1805.02742 [cs.SE]. https://arxiv.org/abs/1805.02742 (2018). Accessed 21 Mar 2019
8. Standart vyshchoi osvity Ukrainy: pershyi (bakalavrskyi) riven, haluz znan 12 – Informatsiini tekhnolohii, spetsialnist 121 – Inzheneriia prohramnoho zabezpechennia (Higher education standard of Ukraine: first (bachelor) level, field of knowledge 12 – Information technologies, specialty 121 – Software engineering). https://mon.gov.ua/storage/app/media/vishcha-osvita/zatverdzeni%20standarty/12/21/121-inzheneriya-programnogo-zabezpechennya-bakalavr.pdf (2018). Accessed 25 Oct 2019
9. Striuk, A.M.: Software engineering: first 50 years of formation and development. In: Kiv, A.E., Semerikov, S.O., Soloviev, V.N., Striuk, A.M. (eds.) Proceedings of the 1st Student Workshop on Computer Science & Software Engineering (CS&SE@SW 2018), Kryvyi Rih, Ukraine, November 30, 2018. CEUR Workshop Proceedings **2292**, 12–36. http://ceur-ws.org/Vol-2292/paper01.pdf (2018). Accessed 21 Mar 2019