Professional competencies of future software engineers in the software design: teaching techniques

Professional competencies of future software engineers in the software design: teaching techniques

A M Striuk[1] **and S O Semerikov**[1,2,3]

[1] Kryvyi Rih National University, 11 Vitalii Matusevych Str., Kryvyi Rih, 50027, Ukraine
[2] Kryvyi Rih State Pedagogical University, 54 Gagarin Ave., Kryvyi Rih, 50086, Ukraine
[3] Institute for Digitalisation of Education of the NAES of Ukraine, 9 M. Berlynskoho Str., Kyiv, 04060, Ukraine

E-mail: andrey.n.stryuk@gmail.com, semerikov@gmail.com

Abstract. The article is devoted to one of the competence components of a mobile-oriented environment for professional and practical training of future software engineers. It is shown that the introduction of higher education standard 121 "Software Engineering" for the first (bachelor) level of higher education in Ukraine has generated a number of training quality assurance problems associated primarily with the low level of detailed competencies and program learning outcomes. By solving these problems, the detailed design of the system of professional competencies for future software engineers is developed. The article deals with the approaches to developing one of the most important special professional competences of future software engineers – the ability to participate in software design, including modeling (formal description) of its structure, behavior, and processes of functioning. Based on a historical and genetic review of the software engineering training practice of future software engineers in the USA, UK, Canada, Australia, New Zealand and Singapore, recommendations for choosing forms of training organization, selection of training content, ways of students' and teachers' activities in software engineering, modeling and designing tools; assessment of the appropriate competence formation level are formulated. The example of organizing design training in conditions close to industrial-studio training is considered. The problems of transition from architectural to detailed design and project implementation are shown. Prospects for further development of this study are to substantiate the third (after requirements engineering and design engineering) engineering component of software engineering – the software construction.

1. Introduction

The approval in 2018 of the higher education standard 121 "Software Engineering" for the first (bachelor) level of higher education [1] in the light of the new accreditation procedure for educational programs in Ukraine gave rise to two main problems, the solution of which was entrusted to the guarantors of educational programs:

(i) The standard provides an extremely high degree of freedom in interpreting the content of competencies and learning program outcomes, which leads to significantly different and non-harmonized educational programs and plans, which are assessed by National Agency for Higher Education Quality Assurance (Ukraine) experts according to the same vague quality criteria. At the same time, similar foreign standards (university, state and world

standards) have a high level of detailing of competence components and criteria for assessing their formation – just as in the USA, Australia and Japan quality criteria for educational programs are detailed for a certain field of education, not applied to all. The lack of clear criteria for quality assessment, specific to the field of knowledge, generates risks of violation of the principle of academic integrity in their evaluation.

(ii) The high rate of changes in the content of the technological component of software engineering training along with the attempt of educational program compilers at least respond to the requirements of industry, if not ahead of them, leads to an instrumental bias in software engineering training – up to the leveling of the universal competences essential for sustainable professional development of a software engineer. Such a bias leads to the indistinguishability of software engineering from other specialties in the branch of knowledge 12 "Information Technology" and professional disorientation of entrants and students.

Solving these problems requires systematic mastering of the international experience in software engineering training in its genesis [2, 3] and designing a system of professional competences (content, indicators of formation and diagnostic tools), both general [4] and special [5].

Among the special competences of an software engineer, those that reflect the "essence and spirit" of software engineering – the specificity of software engineers' professional activity, which cannot be acquired during training in other specialties of the "Information Technologies" branch of knowledge – deserve special attention. Thus, the recommendations for the development of software engineering undergraduate curricula define the competence to find compromises, the essence of which is to reconcile conflicting design goals, to find acceptable compromises in cost constraints, time, knowledge, existing systems and organizations: "Students should engage in exercises that expose them to conflicting and changing requirements. ... Curriculum units should address these issues, with the aim of ensuring high-quality functional and nonfunctional requirements and a feasible software design" [6, p. 21].

Finding compromises and reconciling contradictions is a traditional engineering design activity, which is not mandatory for all information technology professionals, but is key for software engineers. In the standard [1] it corresponds to the special competence K14 – "the ability to participate in software design, including modeling (formal description) of its structure, behavior and functioning processes". The formation of the ability to design software is critical in determining whether a graduate of an software engineering educational program is a software engineer.

A review of software engineering educational programs available at Ukrainian Higher Educational Institutions websites shows that software design training is accomplished in three main ways:

(i) artificial introduction of design elements in different courses in close connection with the tools by the example of solving typical simplified problems of a certain industry;

(ii) spontaneous teaching of design elements during the internship on real tasks;

(iii) training of software modeling tool (usually UML).

Only a small number of educational programs have attempted to take into account the global experience of design training organization based on synergetic combination of academic and industrial forms and methods of training.

Therefore, the purpose of our paper is to provide a historical and genetic review of software design training practices of future software engineers.

2. Teaching software design: international experience

Vladimir N. Pelevin correlates the special competence K14 (ability to participate in software design, including modeling (formal description) of its structure, behavior and functioning

processes) singled out in standard [1] with the ability and readiness to carry out system design, which includes system engineering design, applied software and computer networks [7, p. 14].

Modeling and analysis can be considered the basic concepts of any engineering discipline, as they are important for documenting and evaluating design solutions and alternatives [6, p. 31]. Software design refers to issues, methods, strategies, presentation methods and patterns used to determine how to implement a component or a system [6, p. 32].

David Carrington [8] defines that software design is the stage of software development, during which the specification is transformed into a structure suitable for implementation. Design training should cover both the design object and the process by which this object is created [8, p. 547].

Chenglie Hu [9] puts a number of questions about what software design is:

1. *Can software design be defined using an engineering metaphor?* "For instance, design of a bridge must comply with appropriate laws of physics, but designing software has no laws of any kind to abide by, at least in theory. The architecture of a bridge is much discernible with naked eyes whereas software is intrinsically intangible, and, at core, it is an abstract entity of which we only work with various representations. In practice, an engineering design must precede formal construction of the bridge whereas construction of software can take place without an explicit design. Yet perhaps the most serious difference is that once construction commences, changes to the specification of an engineering product may not be allowed whereas changes to software requirements are generally expected and can indeed happen anytime during the software's life cycle." [9, p. 63].
2. *Can a computer program itself be considered a piece of art?* "It probably can, but not always against the same set of criteria. The reason is that a designer can, for instance, trade violation of some design principles for resolving pressing issues in hand due to software constraints of different nature. Design seeks a balance among extendibility, compliance with design principles, and accommodation of software constraints, and can thus be elegant and artistically appealing in many different ways." [9, p. 63].
3. *Can software design be defined by design activities?* "In fact, design activities are diverse; many are certainly technical, yet some can be social too, but none is likely to be standardized to characterize design or its process." [9, p. 63].
4. *Can software design be defined by design artifacts to be produced or design phases to go across?* "IEEE Standard 1016-2009 (for Information Technology – Systems Design – Software Design Descriptions) specifies the required information content and organization for software design descriptions to be used for communicating design information to its stakeholders. However, the way to describe a design and document information content and organization can range from the more traditional "big design up front" all the way to "the code is my design". When professionals do not agree on design artifacts, neither will they likely have consensus on design phases. In fact, agile practitioners believe that design is not only highly iterative, but emergent, and models often lie. Thus, only coding, running tests, and refactoring the code reveal the truth about a design." [9, pp. 63–64].

Chenglie Hu connects the design process with design thinking: "Professionals have long suspected that designing software might well be a cognitive process that happens inside one's brain at a speed faster than lightening, and the essence of design, then, is a rapid modeling and simulation process that proposes solutions and allows them to fail" [9, p. 64]. Clive L. Dym, Alice M. Agogino, Ozgur Eris, Daniel D. Frey, Larry J. Leifer) listed the mental abilities that are often associated with good project thinkers, including: tolerate ambiguity, maintain sight of the big picture, handle uncertainty, make decisions, think as part of a team, and think and communicate in several languages of design [10, p. 104].

As a result, Chenglie Hu defines software design as a systematic, intelligent process in which designers generate, evaluate, and specify concepts for a software system whose structure and function achieve clients' objectives or users' needs while satisfying a specified set of constraints [9, p. 64].

The design process is both extremely creative and extremely complex, so it is not surprising that there are few truly outstanding designers. However, the demand for new projects is very high, and to face it, each architectural and engineering industry has developed a set of principles that enable designers with average abilities to create design solutions of acceptable quality. Although Software Engineering is much younger than other engineering fields, it has also accumulated some knowledge of how to create projects, and future software engineers need to master this knowledge, so design teaching should be an integral part of their training [11].

"Exceptional individuals are born with a sense of what makes a good design. Most of us must see examples of good design before we can come up with good design on our own. Therefore, we teach design by example, providing students with design sketches and then much feedback on their attempts to refine the design." [12, p. 33]

Robert M. Graham in 1970 in his article [13] proposes a method of teaching design that introduces a single key idea and explores its implications in an environment that consciously ignores the issue of efficiency. Once the logical implications of this idea have been fully developed, the additional key ideas are combined one by one and their implications are explored. Next, the effectiveness and other real limitations are considered, examining the consequences of each. This procedure is continued until a realistic project is reached. In addition, a specially designed and documented case is used to illustrate project development.

David M. Weiss in [14] presents a later (mid-1980s) experience of design software teaching invariant to the applied design methodology. His approach involves two stages: first, students are introduced to the principles underlying the methodology and the experience of applying these principles to small, well-defined tasks. In the second stage, students are offered a real task that they must solve independently in a simulated work environment, receiving guidance only on methodological issues. The first stage is presented in the form of a series of lectures. The second is supervised by a design expert. As a result of the first stage, students gain some understanding of the principles underlying the design methodology, see examples of application of these principles and practice their application on educational examples. After the first stage, students are able to use design techniques under the guidance of an experienced designer in the process of his almost daily interaction with students. As a result of the second stage, students gain enough experience in design to carry it out with less guidance, communicating with an experienced designer about once a week [14, p. 1156].

At the kick off meeting of the project, the roles they will play during the project are distributed among the students. Students get acquainted with the sequence of the main stages of the project, get a list of documents that have to be presented, and the procedure for their preparation. Students are also provided with brief notes on the configuration of management procedure, a description of the responsibilities and the composition of the quality assurance team. Each student must develop and document at least one module of the system [14, p. 1157]. D. M. Weiss notes that students were dissatisfied with the fact that they focused on design, but did not develop the finished product due to the complexity of the design task (however, a simpler task would not provide an opportunity to demonstrate and master all necessary design techniques and methods). In general, the students felt that they had mastered the design methodology, but were disappointed that they did not create a single line of code. To overcome this disappointment, students can be invited to continue working on the project in the next semester [14, p. 1158].

D.M. Weiss provides the following recommendations for the organization of software design training [14, p. 1159]:

(i) students must be advised by a design specialist who can ensure strict adherence to the

methodology;

(ii) the design task must be quite complex, and students are not allowed to simplify it;

(iii) students should be encouraged to make their own project decisions: none of the teachers should interfere in the decision-making process;

(iv) the size of the student group should be small to allow teachers to closely monitor the progress of each student.

Ehud Lamm [15] notes that the goal of a software design course is to develop software design skills by teaching the basic concepts of software engineering needed to study and analyze alternative software projects.

Computing Curricula 2020 [16, p. 120] defines the following competencies related to software design:

1. Present to business decision-makers architecturally significant requirements from a software requirements specification document.
2. Evaluate and compare tradeoffs from alternative design possibilities for satisfying functional and non-functional requirements and write a brief proposal summarizing key conclusions for a client.
3. Produce a high-level design of specific subsystems that is presentable to a non-computing audience by considering architectural and design patterns.
4. Produce detailed designs for a client for specific subsystem high-level designs by using design principles and cross-cutting aspects to satisfy functional and non-functional requirements.
5. Evaluate software testing consideration of quality attributes in the design of subsystems and modules for a developer/manufacturer.
6. Create software design documents that communicate effectively to software design clients such as analysts, implementers, test planners, or maintainers.

The core of knowledge about software design Chenglie Hu [9] proposes to divide into 2 categories: design process knowledge and knowledge of design-enabling techniques – the latter includes design patterns. Knowledge of the process is defined by the author as "knowledge about commonly-recognized design phases we go through as well as paradigms or methodologies we use to progress in a design process or to produce design artifacts" [9, p. 65]. Analysis of software requirements is part of the architectural design process, which covers the functional and behavioral aspects of architecture. An architectural design concerns not only what the system is, but also what the system does. The next stage of design – non-architectural (detailed) design – is to modulate and detail the interfaces of design elements, their algorithms and procedures, as well as the types of data needed to support the architecture and meet the requirements. Non-architectural design determines the functionality and structure of the software at a high level of detail, but insufficient for implementation. The design process also involves managing the production of design artifacts to ensure the correct and accurate implementation of design ideas and solutions. Chenglie Hu believes that the design process is not complete if the project is not implemented in the code [9, p. 66].

Keith Pierce, Linda Deneen, Gary Shute [11] believe that the key issue of design training is a rational choice among many alternative solutions [11, p. 220]. "It seems to us that instruction in design would improve dramatically by making this single modification in how we teach: presenting alternative design strategies and alternative solutions derived using these strategies, presenting the criteria for judging the quality of the alternatives, and forcing students in laboratory exercises to make and justify a choice among them." [11, pp. 220–221] The authors divide the design methods into low-level (used to design small modules – the choice of ordered and unordered arrays, between recursion and iteration, etc.) and high-level (used to decide on the organization of software modules).

Yanxia Jia and Yonglei Tao [17] consider modeling to be a central component of the quality software development process. When teaching software design, teachers should emphasize the creation of an appropriate model to consider the problem and the use of model properties to design a solution [17, p. 702].

The authors [17] identify the following key concepts:

- *modeling* – the process of creating an abstract, graphic or mathematical description of the design problem, during which the developer replaces a complex and detailed real situation with an understandable model that reflects the essence of the problem;
- *the evolution of the model* in the iterative process of development tends to preserve the form of representation, while *the transformation of the model* involves a change in the point of view from which the design problem is considered, and a change in the structure of the design model;
- *code refactoring* is the process of changing the code of a computer program in order to preserve its ability to develop, improve its readability or simplify the structure, while maintaining existing functionality. Refactoring training not only gives students practical programming skills, but also helps them understand the most important principles of Software Engineering;
- *design patterns* can help developers solve certain design problems, as well as improve existing projects.

Redesigning software to better reuse or acquire other qualities is an illustration of model transformation. The gradual transformation of the model requires students to constantly assess the gap between what has been done and what needs to be done. Such activities are especially useful for the formation of students' ability to evaluate the existing solution and analyze the benefits and costs [17, p. 705].

Software design patterns are often used to solve a common problem through a "general" approach to it. Johan van Niekerk and Lynn Futcher [18] point out that the design pattern provides a conceptual model of the best practice solution, which in turn is used by developers to create a concrete implementation of their task. The use of design patterns has a number of advantages: 1) the pattern provides a guide to best practice; 2) the use of the pattern provides developers with a common "dictionary" for easy and clear discussion of complex design concepts. Due to these and other advantages, design patterns are often studied in software design courses [18, p. 75].

A pattern can be described as "a solution to a problem in a context" [18, p. 77]:

- the context is a recurring situation in which a pattern is used;
- the problem refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context;
- the problem should be a recurring problem;
- the solution provides a general design (core solution) that extracts the essence of the solution to resolve the problem for the given context and constraints.

Design patterns provide software designers with three main advantages [18, p. 78]:

(i) the solution is known to be sound because it is time-tested;
(ii) benefits and drawbacks of a pattern are known in advance and they can be taken into account while sketching the solution;
(iii) patterns establish a common vocabulary that can ease communication between different stakeholders.

Chad Williams and Stan Kurkovsky [19] emphasize that the process of learning to use appropriate software design patterns can be made creative by transforming the design process itself into a constructivist learning environment. To do this, they propose not to limit the subject of student projects and create conditions for students to get out of the comfort zone by applying new hardware platforms and interfaces to them. The interim evaluation of the project was performed according to the level of expediency and the number of used design patterns, and the final - according to the level of creativity of the whole project.

A framework can be defined as a semi-complete program that contains certain fixed components that are common to all programs in the problem area, along with certain variable components that are unique to each program created from it. Most commercial software is developed using frameworks by extending and customizing the standard common features they provide. Zoya Ali, Joseph Bolinger, Michael Herold, Thomas Lynch, Jay Ramanathan and R. Rajiv Ramnath [20] indicate that developers who know the principles of object-oriented design do not use in the development of software a particular framework, focusing simply to "make it work". Adapting to a new framework can be a challenge for novice developers because using design patterns in a new framework can lead to poor design and misuse of the framework. The authors [20] propose a three-step process of learning design using frameworks, aimed at overcoming this problem:

(i) First, students are asked to design their program using object-oriented design. Using the formulation of the problem, students are encouraged to "object-oriented thinking" and use of previous experience to create objects, classes, responsibilities, relationships, methods, and other UML entities.

(ii) Next, students are asked to rewrite the program using design patterns. One of the patterns is the Model-View-Controller (MVC) pattern. The idea of this model is to isolate the logic of the program domain from the way the data is presented to the user (ie the user interface of the program) so that these two very important components of any program can be developed, implemented and maintained separately.

(iii) Finally, students are asked to adjust the use of the design pattern according to the chosen framework.

To better understand the framework and seamlessly integrate the project with the framework, students need to learn the design patterns dictated by the framework and compare them with the standard design pattern. "Similarly, the UML is a natural complement to design patterns, providing a means for students and teachers to communicate them." [21, p. 42]

Teaching UML encourages a downward approach to software engineering: first, students are taught to identify software requirements, then to transform requirements into software architecture, low-level design, and finally implementation. UML is often used as the "lingua franca" across these lifecycle phases and heavy emphasis is placed on the students producing high quality, syntactically and semantically correct UML diagrams [22, p. 19].

Ali et al. [20] offer a methodology that will give students the opportunity to take advantage of the framework in the implementation of their project [20, pp. S3G-3–S3G-4]:

1. Paraphrase the problem statement and extract all the nouns and verbs from it. The nouns serve as candidate objects, classes and attributes, while verbs serve as responsibilities.
2. Merge the extracted nouns into classes. This may require discarding irrelevant nouns or nouns representing the same thing.
3. Merge extracted verbs into classes, instances and responsibilities.
4. Assign responsibilities by identifying required methods to complete those responsibilities.
5. Walk through the scenario to ensure that each scenario is supported by methods and identify the collaborations between them.

Design is inherently an interdisciplinary subject, influenced by a large number of human factors, so teaching design is not only a process of teaching design itself, but also a process of further developing students' social skills [9, p. 70]. Stanislaw Jarzabek notes that team-oriented project courses provide an opportunity to learn the principles of Software Engineering when their application is really necessary and profitable, and suggests the following classification [12, pp. 31–32]:

(1) Industrial attachments in which students work on real-world problems in industrial settings.
(2) Project courses in which students work on problems in various application domains under supervision of faculty members, experts in a given domain. Sometimes such courses build on real-world problems provided by industry.
(3) Project courses in which students learn advanced software design principles and apply them in their projects. As faculty members need scrutinize in detail design artifacts to provide feedback to students, such projects must be supervised by faculty members specializing in software engineering, and well-versed with problem and solution domain students work with.
(4) Projects developed from scratch versus projects in which students extend existing software.
(5) Projects based on a specific software platform such as .NET, JEE, service, mobile device or Facebook.

"Team work, communication and writing skills can be trained in all of the above project courses. Other skills are quite difficult to accommodate in the frame of a single project course. For example, in project types (1) and (2) the goal is to expose students to the reality of fuzzy, ill-defined and changing requirements. Fuzzy requirements and software design are not only the two hallmarks of software development, but also hard and wicked problems that are difficult to teach in the frame of a single course. Project courses that expose students to fuzzy and changing requirements tend to be less structured and rigorous than courses that teach students application of design principles. When teaching application of design principles (project course type (3)), we should give students sample design sketches, and lots of detailed feedback on their initial attempts to refine the design. Supervisors need be intimately familiar with a problem domain and design solutions to provide effective guidance for students. This may be quite difficult in projects types (1) and (2)." [12, pp. 32]

Students write a final report in which they document project plans, development process, architectural and detailed (non-architectural) design decisions. Each team is given one hour to present their work. Teachers evaluate students on the basis of design quality, ability to evaluate design decisions and justify their choice in view of the stated attributes of quality - reuse, extensibility and effectiveness of the query evaluation strategy [12, pp. 37–38].

Learning by way of active feedback is an effective way to teach students to design on a large scale. Examples provided to students may include software architecture sketches, API specifications, and illustrations of how to apply design techniques. Some students clearly follow a specific example to create their own design solutions, while others learn from examples, but then innovate, experiment with ideas, and offer their own design techniques. In both cases, it is important that students clearly understand the essence of the design methodology. Although the lectures highlight the theory of design principles, communication between teams and teachers leads to their understanding. At the beginning of their studies, students are especially often mistaken and need a lot of feedback and constructive discussions to perform proper design [12, p. 38].

Damian A. Tamburri, Maryam Razavian and Patricia Lago [23] noted that this approach helps students in learning the main challenges of software design [23, pp. 61–62]:

(a) *accountable and rational design decisions* – students learn how to reason and really be accountable for their own design decisions;

(b) *collaborative design* – students can brainstorm constructively with "opponent" teams, reaching a deeper understanding of the designer role;

(c) *iterative design* – students learn from other people's mistakes and solutions pro-actively revisiting their own design, which is far more effective than any of the many examples we used in the previous years;

(d) *"social" design* – students learn to make teamwork effective and cope with critical reviews driven by different backgrounds and expertise (hence offering feedback from very different perspectives).

J. L. Murtagh and J. A. Hamilton Jr. [24] describe the organization of project-oriented learning, determining the following desired learning outcomes of students:

(i) Students should use the knowledge of previous computer science courses to develop moderately complex computer projects, based on clear, consistent and reasonably complete requirements presented by the teacher in the project task.

(ii) Students must develop high-level (architectural) projects of programs and their interfaces and obtain the approval of the teacher before moving on to detailed design. Students must demonstrate that all requirements for the project assignment have been allocated to specific parts of the architectural design.

(iii) Students must develop a detailed (non-architectural) project of software and all interfaces and obtain the approval of the teacher before embarking on implementation (ie writing code).

(iv) Students should develop a test plan for each project. The test plan should demonstrate how all software requirements will be tested, and include a software testing schedule.

(v) Students should develop documentation that shows how their software and test plan meet the requirements of IEEE / EIA 12207 [24, pp. 5.577.2–5.577.3].

"There is an interesting parallel between teaching and learning on the one hand and design patterns on the other. Design patterns are little more than good practice; they are the culmination of tried and tested techniques for designing software that exhibits desirable properties like flexibility and reuse. On several occasions, undoubtedly along with other software designers, we have solved a design problem only to later find that what we have actually done has been to apply a particular design pattern. This is reassuring, but the point is that we have subconsciously applied what is accepted to be good practice. With teaching, we often do the same; we unknowingly use a technique that has its roots in established education theory. In both cases, however, adopting proven techniques is important to yield quality results." [21, p. 40]

Ian Warren [21] methodically substantiated the learning outcomes of software design training [21, pp. 41-42]:

1. *Identify and describe the objectives of software design.* Design objectives include correctness, robustness, flexibility, reusability and efficiency. Students should appreciate that software should not only be correct, but that the latter 4 non-functional objectives are also important.
2. *Interpret and construct UML models of software.* The UML, being a standard industry notation, is an obvious choice for communicating design knowledge. Essentially, students should be able to read and write UML models.
3. *Explain the notion of design patterns and describe a subset of patterns.* Design patterns embody proven design solutions. Students should appreciate that using patterns fosters an engineering approach as opposed to one that solves problems from first principles.
4. *Apply patterns to solve real-world problems, making sensible tradeoffs where necessary.* Awareness of patterns is important but is no substitute for being able to apply a pattern to solve a problem. This objective is concerned with deeper, functioning knowledge.

5. *Apply newly acquired and developed programming skills.* Students have functioning knowledge of fundamental OOP but have limited knowledge of Java class libraries and more advanced aspects of programming. This objective aims to equip students with stronger implementation skills.
6. *Work with relatively large software projects.* Similarly, students' experience of developing software is typically confined to small programs involving a few classes. Exposing students to larger projects is good preparation for final year project work and industrial practice.

I. Warren singled out methods that are effective for teaching software design:

- *problematic questions such as "what will happen if ...?"*. I. Warren gives an example of the use of such questions when considering UML class diagrams and, in particular, relationships and multiple constraints: "By presenting a class diagram and asking students what the multiplicity constraints really mean engages students; they have to interpret the diagrams and assess their own understanding rather than sitting passively where it is too easy for students to think they have understood. When we see that students have understood, we can change the constraints; subtle changes on a diagram can have significant changes in meaning." [21, p. 43];
- *role-playing games*: "In reviewing the basic ideas of object-orientation we have students acting as objects and playing out scenarios showing how links between objects are formed dynamically and how messages are processed. ... Role play leads naturally into documenting scenarios using UML; once students have exercised a scenario, they document it using a UML object interaction diagram. From an initial class diagram and an interaction diagram generated from role play, we are able to explore the issue of well-formed models, where different views should ultimately be mutually consistent." [21, p. 43];
- *active learning*, from the introduction of activity fragments during lectures to sessions of interactive and problem-oriented activities as a means of acquiring functional knowledge by students about design patterns [21, p. 48];
- *peer-learning* "is manifested by students having a formal learning partner who they pair-program with, and with who they collaborate on coursework tasks. In addition, students work in small groups on problems in the classroom." [21, p. 48].

Jon Whittle, Christopher N. Bull, Jaejoon Lee, and Gerald Kotonya [22] offer another alternative to traditional learning, studio-based education, which brings to the forefront reflective practice as a way to develop design skills. In the studio, students are invited to critically reflect on their own and others' design, using a variety of techniques such as mentoring, design criticism, mutual mentoring, and concerted evaluation. The studio course is usually, but not always, taught in a room designed for this purpose, ie in the studio. Physical studio is considered a key element because it allows students to constantly demonstrate their own work, which over time encourages reflective practice. The studio also encourages frequent and informal interactions between students, which leads to mutual learning [22, p. 12].

In the table 1 a comparative description of the environment of studio and traditional learning is shown.

When pursuing a studio course at the University of Lancaster, the authors [22] found that students did not use formal modeling (modeling using formalized descriptions, such as UML, where exact notation and correct syntax are used), using informal modeling using or special notation (for example, drawing figures on a board (figure 1), pen on paper or using a digital sketching tool), or free interpretation of a recognized modeling language (for example, drawing a diagram of UML classes without proper syntax) [22, p. 16]. Although the students in the studio did little formal modeling, they regularly came up with informal models. These include both UML models (typically usage variants and class diagrams), sketches (eg. high-level architecture,

Table 1. Comparative characteristics of studio and traditional learning (according [22, p. 15]).

Aspect	Studio course	Traditional course	Example: Lancaster University's study course
Physical environment	There is a physical room (i.e., the studio) which is open and reconfigurable providing a variety of group, individual and social spaces	Standard lab	Dedicated lab with 24 hour access, maintained by students themselves
Management of studio	Rules regarding use of the space should not be restrictive	Lab tightly controlled by University	24 hour access; food/drink allowed; students have admin rights
Modes of education	Teaching staff play a coaching/mentoring role rather than being didactic	Students given a prescriptive list of documentation to produce	Students were told to produce "as much documentation as needed"; there was no prescription on what kinds of diagrams or notations to use
Awareness	Placing work on display (as works-in-progress or final products); visibility of work helps students see each other's work.	No special consideration	Students used mobile whiteboards in the studio to display design work, which was left up for the duration of the project
Critique	Ongoing critique is used for providing feedback and developing ideas. It should take place in multiple ways (formal and informal, group and individual, peer and staff)	Provided only as part of weekly meetings with project supervisor	Provided on a continuous basis using a variety of methods: individual and group demos/presentations, informal coaching, peer critique, critique from external assessors (e.g., companies) and formal judging
Culture	A studio culture should be social and foster sharing, and yet should be sensitive to sup- porting a good work ethic	Lack of a dedicated lab meant that students typically only met at prearranged times	Students used the studio as a home, leading to serendipitous interactions and a feeling of 'belonging' to a cohort
Inspiration	When designing, students should be encouraged to be creative in their designs and solutions	Project specification decided a priori by academic staff	Students come up with their own project ideas in a facilitated creativity brainstorming session

user interface design sketches), and process-based models (eg. burnout diagrams used in flexible design) [22, p. 17].

In the studio approach, students engaged in modeling as much as needed and when they needed it. The students did not try to create fully-fledged models for the function they were

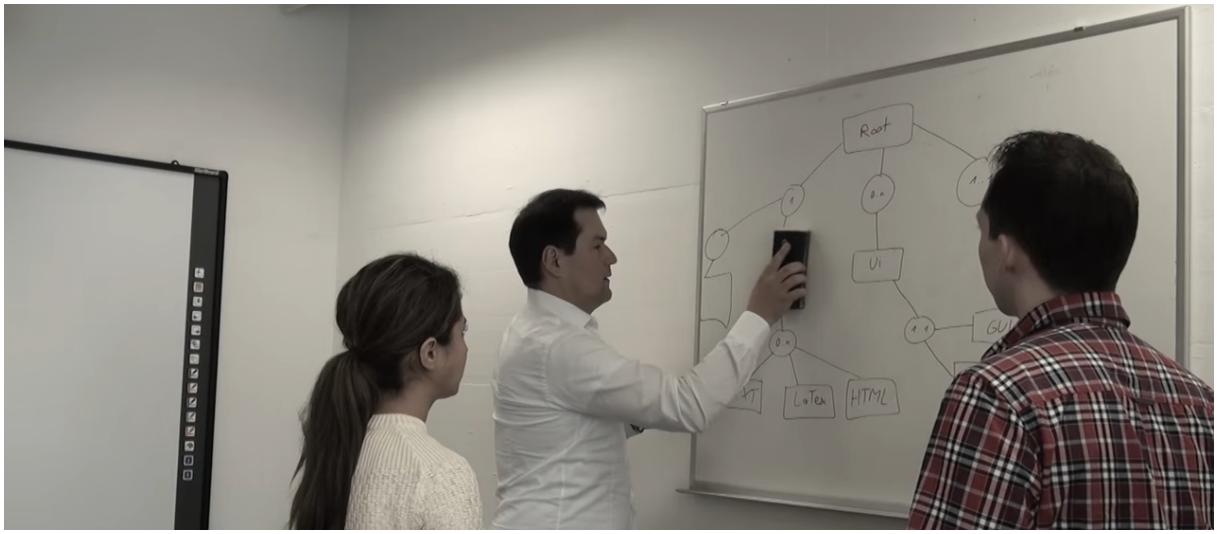

Figure 1. Informal modeling on the board.

going to implement – the models were used mainly for brainstorming and design. Once the design was thought out enough to allow the team to move on to the next stage of development, the models did not change, but continued to be used as a design artifact: students typically used them either to remind them of what they were doing, or as a background "noise", which in some way helped them work in a group [22, p. 18]. This approach is actively used by software design experts – for example, Marian Petre and André van der Hoek in the guide [25] used informal modeling to summarize their own design experience (figure 2).

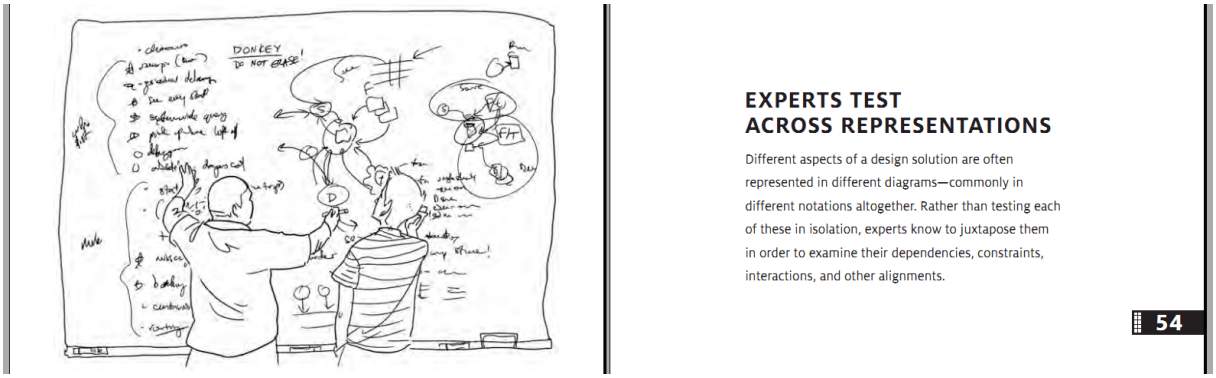

Figure 2. Fragment of the design guide of M. Petre and A. van der Hook [25] in the form of a sketch project.

The combination of informal and formal modeling is possible with the help of flexible modeling tools (figure 3), which after creating an informal sketch provide an opportunity to move to formal modeling without changing the tool [22, p. 17]. The authors of the studio course [22] believe that students should be introduced to a number of design methods, learn to apply these methods in practice and through reflective practice to encourage them to learn about the pros and cons of these methods: "Indeed, one could argue that the most important thing to teach is the culture of reflective practice itself. If students learn how to reflect, they will become reflective practitioners and can apply those skills to any new method or approach which they are faced with in their future careers." [22, p. 20]

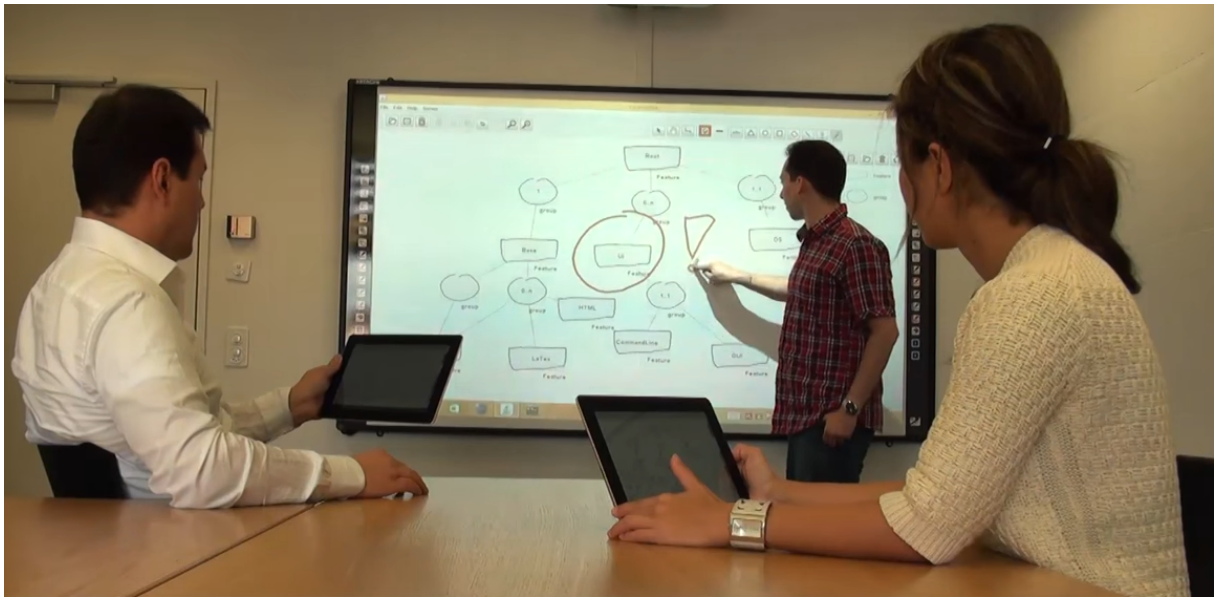

Figure 3. Simulation using FlexiSketch Team.

M. Petre and A. van der Hook [25, pp. 148–158] recommend the following actions for reflection in software design:

(i) Do not allow deviations to minor issues when discussing the project, regularly check yourself, "where am I and what am I doing?"

(ii) Based on a deep understanding of the fundamental concepts of design "keep up to date": remember what design decisions have already been made, why they were so and what other decisions need to be made.

(iii) Thinking about what is not designed: identifying and considering design limitations, to identify the design excessively or insufficiently.

(iv) Periodically stop to look at the project as a whole, asking yourself whether the goals of customers, user perceptions, the market itself, etc. have changed.

(v) Anticipate different options for the future, analyzing the economic feasibility of determining where to invest – in methods, tools, resources, design alternatives – in order to save efforts in the future.

In the study of design Chenglie Hu recommends to consider the following [9, pp. 69-70]:

1. Students' technical competency and cognitive strengths can significantly impact the outcomes of learning design. For this reason, the goal of teaching software design is not to make every student a designer, but for students to experience what it might take to produce a good design while acquiring individually achievable design skill and ability by seeking pedagogy that can maximize learner's technical and cognitive potential.
2. For relatively small scale design problems, test-driven development renders opportunities to express modeling ideas directly in code, allowing more effective evaluation of design tradeoffs and committing less design errors than using diagrams.
3. Learning "small-scale design" (writing related methods, appropriate data encapsulation, using good method names, etc.) is probably much less challenging than studying design in general to study structural stability, which requires design compromises to apply knowledge of the design process. The minimum basic design skills (with an appropriate level of project

thinking) that graduates must acquire may include the ability to effectively begin design, decompose a task, perform design iterations with sound design compromises solutions, and implement the project in code.

4. Students cannot effectively acquire the ability to design and learn project thinking if they do not solve design problems of appropriate complexity.
5. Despite the lack of universal formulas or design recipes, data analysis (identification of entities and their relationships) and process analysis (detection of actions and logical flows) are critical to understand what information to use and how it is transformed.
6. Correctly documenting a design – timely, not post factum – is a necessary skill, however, whether students should use UML or how accurately they use the language to document the model is not so important. It is advisable to acquaint students with some classic diagrams, encouraging their creative use.
7. For the final assessment of students' learning outcomes, it is advisable to combine traditional classroom tests (to make sure "they know this") with complex design homework, which must be solved individually or in teams of two with careful monitoring and evaluation of finished design products made by students, but also the quality of project implementation, documented in student reports.

3. Conclusions

Summarizing the results of the study has provided an opportunity to draw the following conclusions:

1. Software design is a type of engineering design, which combines two components – creative and systemic-technical. The formation and development of engineering creativity is a technology for mastering the best examples of projects in the process of design activities close to production. The result of the design is an updated and adapted sample (typical project, or design pattern) or an original new design (project). In this regard, significant potential for the development of methods for teaching software design to future software engineers is available in related research in other fields of engineering (including construction, mechanical and computer) and art (in particular, architecture, painting).
2. In teaching software design, special attention should be paid to supporting students in design, the theoretical foundations of which are acquired in the form of lectures, in the form of practical classes, studies, course projects, etc. to solve real problems of complex software design. This places an additional requirement on design teachers to work in software development industry or to be closely connected with its customers.
3. Software modeling plays a key role in its successful design. At the same time, compliance with the requirements for the use of accurate formal descriptions of models in the design process does not significantly affect the quality of design: the ease of use of accurate formal and flexible informal models is comparable. Therefore, it is advisable to use flexible methods and tools of modeling, in which the construction of formal UML diagrams is performed on an informal sketch project.
4. The transition from architectural design based on the chosen pattern to non-architectural (detailed) design may require adapting the pattern to non-design realities in the form of software design tools, such as the framework. Therefore, despite the fact that software design is possible without its design, the appropriate design result is at least a constructed software prototype: as part of the software life cycle, design is the driving force of its development – it is constantly performed until decommissioning.
5. Assessment of the formation of competence in software design involves testing individual knowledge and team skills of design in the process of solving a complex design problem.

Design artifacts such as sketch designs, formal models, design documentation, prototypes and more are important for this. Therefore, the defense of projects is an appropriate form of evaluation.

Prospects for further development of this study are to substantiate the third (after requirements engineering and design engineering) engineering component of software engineering – *software construction.*

ORCID iDs
A M Striuk https://orcid.org/0000-0001-9240-1976
S O Semerikov https://orcid.org/0000-0003-0789-0272

References

[1] Ministerstvo osvity i nauky Ukrainy 2018 Pro zatverdzhennia standartu vyshchoi osvity za spetsialnistiu 121 "Inzheneriia prohramnoho zabezpechennia" dlia pershoho (bakalavrskoho) rivnia vyshchoi osvity (On approval of the standard of higher education in the specialty 121 "Software Engineering" for the first (bachelor's) level of higher education) URL https://mon.gov.ua/storage/app/media/vishcha-osvita/zatverdzeni%20standarty/12/21/121-inzheneriya-programnogo-zabezpechennya-bakalavr.pdf
[2] Striuk A M 2018 *CEUR Workshop Proceedings* **2292** 11–36 ISSN 16130073
[3] Striuk A M and Semerikov S O 2019 *CEUR Workshop Proceedings* **2546** 35–57 ISSN 16130073 URL http://ceur-ws.org/Vol-2546/paper02.pdf
[4] Semerikov S, Striuk A, Striuk L, Striuk M and Shalatska H 2020 *E3S Web of Conferences* **166** 10036 URL https://doi.org/10.1051/e3sconf/202016610036
[5] Striuk A M, Semerikov S O, Shalatska H M and Holiver V P 2022 *CEUR Workshop Proceedings* 3–11
[6] Joint Task Force on Computing Curricula, IEEE Computer Society and Association for Computing Machinery 2015 *Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* A Volume of the Computing Curricula Series URL https://www.acm.org/binaries/content/assets/education/se2014.pdf
[7] Pelevin V N 2010 *Formirovanie professionalnoi kompetentnosti budushchikh bakalavrov po napravleniiu "Informatcionnye sistemy i tekhnologii" (Formation of professional competence of future bachelors in the direction of "Information systems and technologies")* Dissertation for degree of candidate of pedagogical sciences: 13.00.08 – theory and methods of vocational education Ural State Technical University – UPI named after the first President of Russia B. N. Yeltsin URL https://elar.rsvpu.ru/handle/123456789/987
[8] Carrington D 1998 Teaching software design and testing *FIE '98. 28th Annual Frontiers in Education Conference. Moving from 'Teacher-Centered' to 'Learner-Centered' Education. Conference Proceedings (Cat. No.98CH36214)* vol 2 pp 547–550 vol.2 URL https://doi.org/10.1109/FIE.1998.738732
[9] Hu C 2013 *ACM Inroads* **4** 62–72 ISSN 2153-2184 URL https://doi.org/10.1145/2465085.2465103
[10] Dym C L, Agogino A M, Eris O, Frey D D and Leifer L J 2005 *Journal of Engineering Education* **94** 103–120 URL https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2168-9830.2005.tb00832.x
[11] Pierce K, Deneen L and Shute G 1991 Teaching software design in the freshman year *Software Engineering Education* ed Tomayko J E (Berlin, Heidelberg: Springer Berlin Heidelberg) pp 219–231 ISBN 978-3-540-38418-2 URL https://doi.org/10.1007/BFb0024294
[12] Jarzabek S 2013 Teaching advanced software design in team-based project course *2013 26th International Conference on Software Engineering Education and Training (CSEE T)* pp 31–40 URL https://doi.org/10.1109/CSEET.2013.6595234
[13] Graham R M 1970 *SIGCSE Bull.* **2** 56–60 ISSN 0097-8418 URL https://doi.org/10.1145/873641.873652
[14] Weiss D M 1987 *IEEE Transactions on Software Engineering* **SE-13** 1156–1163 URL https://doi.org/10.1109/TSE.1987.232864
[15] Lamm E 2003 Booch's Ada vs. Liskov's Java: Two approaches to teaching software design *Reliable Software Technologies — Ada-Europe 2003* ed Rosen J P and Strohmeier A (Berlin, Heidelberg: Springer Berlin Heidelberg) pp 102–112 ISBN 978-3-540-44947-8 URL https://doi.org/10.1007/3-540-44947-7_7
[16] CC2020 Task Force 2020 *Computing Curricula 2020: Paradigms for Global Computing Education* (New York, NY, USA: Association for Computing Machinery) ISBN 9781450390590 URL https://doi.org/10.1145/3467967

[17] Jia Y and Tao Y 2009 Teaching software design using a case study on model transformation *2009 Sixth International Conference on Information Technology: New Generations* pp 702–706 URL https://doi.org/10.1109/ITNG.2009.114

[18] van Niekerk J and Futcher L 2015 The use of software design patterns to teach secure software design: An integrated approach *Information Security Education Across the Curriculum* ed Bishop M, Miloslavskaya N and Theocharidou M (Cham: Springer International Publishing) pp 75–83 ISBN 978-3-319-18500-2 URL https://doi.org/10.1007/978-3-319-18500-2_7

[19] Williams C and Kurkovsky S 2017 Raspberry Pi creativity: A student-driven approach to teaching software design patterns *2017 IEEE Frontiers in Education Conference (FIE)* pp 1–9 URL https://doi.org/10.1109/FIE.2017.8190735

[20] Ali Z, Bolinger J, Herold M, Lynch T, Ramanathan J and Ramnath R 2011 Teaching object-oriented software design within the context of software frameworks *2011 Frontiers in Education Conference (FIE)* pp S3G–1–S3G–5 URL https://doi.org/10.1109/FIE.2011.6142889

[21] Warren I 2005 Teaching patterns and software design *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42* ACE '05 (AUS: Australian Computer Society, Inc.) p 39–49 ISBN 1920682244 URL https://doi.org/10.5555/1082424.1082430

[22] Whittle J, Bull C N, Lee J and Kotonya G 2014 *CEUR Workshop Proceedings* **1346** 12–21 URL http://ceur-ws.org/Vol-1346/edusymp2014_paper_1.pdf

[23] Tamburri D A, Razavian M and Lago P 2013 Teaching software design with social engagement *2013 26th International Conference on Software Engineering Education and Training (CSEE T)* pp 61–69 URL https://doi.org/10.1109/CSEET.2013.6595237

[24] Hamilton, Jr J A and Murtagh J L 2000 Teaching a real world software design approach within an academic environment *2000 Annual Conference* 10.18260/1-2–8739 (St. Louis, Missouri: ASEE Conferences) p 5.577.1–5.577.8 https://peer.asee.org/8739

[25] Petre M and van der Hoek A 2016 *Software Design Decoded: 66 Ways Experts Think* (Cambridge: The MIT Press)