2020

Novice Learner Experiences in Software Development: A Study of Freshman Undergraduates

Catherine Higgins

Ciaran O'Leary

Claire McAvinia

See next page for additional authors

Authors
Catherine Higgins, Ciaran O'Leary, Claire McAvinia, and Barry J. Ryan

Novice Learner Experiences in Software Development: A Study of Freshman Undergraduates

Catherine Higgins(✉), Ciaran O'Leary, Claire McAvinia, and Barry Ryan

Technological University Dublin, Dublin, Ireland
catherine.higgins@tudublin.ie

Abstract. This paper presents a study that is part of a larger research project aimed at addressing the gap in the provision of educational software development processes for freshman, novice undergraduate learners, to improve proficiency levels. With the aim of understanding how such learners problem solve in software development in the absence of a formal process, this case study examines the experiences and depth of learning acquired by a sample set of novice undergraduates. A novel adaption of the Kirkpatrick framework known as AKM-SOLO is used to frame the evaluation. The study finds that without the scaffolding of an appropriate structured development process tailored to novices, students are in danger of failing to engage with the problem solving skills necessary for software development, particularly the skill of designing solutions prior to coding. It also finds that this lack of engagement directly impacts their affective state on the course and continues to negatively impact their proficiency and affective state in the second year of their studies leading to just under half of students surveyed being unsure if they wish to pursue a career in software development when they graduate.

Keywords: Software development undergraduate education · Freshman university learners · Kirkpatrick framework · SOLO taxonomy

1 Introduction

The rapid growth in technologies has increased the demand for skilled software developers and this demand is increasing on a global scale. A report from the United States Department of Labor [1] states that employment in the computing industry is expected to grow by 12% from 2014 to 2024; a higher statistic than the average for other industries. However, learning how to develop software solutions is not trivial due to the high cognitive load it puts on novice learners. Novices must master a variety of skills such as requirements analysis, learning syntax, understanding and applying computational constructs and writing algorithms [2]. This high cognitive load means that many novice developers focus on programming language syntax and programming concepts and, as a result, find the extra cognitive load of problem solving difficult [3]. This suggests

H. C. Lane et al. (Eds.): CSEDU 2019, CCIS 1220, pp. 308–330, 2020.
https://doi.org/10.1007/978-3-030-58459-7_15

that there is a need for an educational software development process aimed at cognitively supporting students in their acquisition of problem solving skills when developing software solutions. However, even though there are many formal software development processes available for experienced developers, very little research has been carried out on developing appropriate processes for freshman, university learners [4]. This lack of appropriate software development processes presents a vacuum for educators with the consequence that the skills required for solving computational problems–specifically carrying out software analysis and design - are typically taught very informally and implicitly on introductory courses at university [5, 6]. This is problematic for students as without systematic guidance, many novices may adopt maladaptive cognitive practices in software development. Examples of such practices include rushing to code solutions with no analysis or design; and coding by rote learning [7]. These practices can be very difficult to unlearn and can ultimately prohibit student progression in the acquisition of software development skills [7, 8]. It has also been found that problems in designing software solutions can persist even past graduation [9].

To address these challenges, this paper presents results and findings from a focused case study which is the first part of a larger research project, the ultimate aim of which is to develop an educational software development process with an associated tool for novice university learners. An adaption of the Kirkpatrick evaluation model [10] is used to frame the evaluation in this study.

In a companion paper which first presented results from this study [11], the findings from the application of the first two levels of the adapted evaluation model (known as AKM-SOLO) were presented. In this extended paper, a detailed description of the full structure and application of the AKM-SOLO model is included. Furthermore, a summary set of results from the first two levels and full results from the remaining levels of the adapted model are presented and discussed. The aim of the study is to identify the specific issues and behaviour that can arise in the absence of a software development process when instructing novice undergraduate learners.

2 Related Research

There has been a wealth of research over three decades into the teaching and learning of software development to improve retention and exam success rates at university level. Research to date has focused on a variety of areas such as reviewing the choice of programming languages and paradigms suitable for novice learners. A wide variety of languages have been suggested from commercial to textual languages through to visual block-based languages [12]. Other prominent research has included the development of visualisation tools to create a diagrammatic overview of the notional machine as a user traces through programs and algorithms [13, 14]; and the use of game based learning as a basis for learning programming and game construction [15, 16].

Research that specifically looks at software development practices for introductory software development courses at university level have tended towards the acquisition of programming skills, with the focus on analysis and design skills being studied as part of software engineering courses in later years. Examples of such research include Dahiya [17] who presents a study of teaching software engineering using an applied

approach to postgraduate and undergraduates with development experience, Savi and co-workers [18] who describe a model to assess the use of gaming as a mechanism to teach software engineering and Rodriguez [19] who examines how to teach a formal software development methodology to students with development experience.

In examining research into software development processes aimed at introductory courses at university, comparatively few were found in the literature. Those that have been developed tend to focus on a particular stage of the development process or on a development paradigm. Examples include the STREAM process [4] which focus on design in an object oriented environment; the P^3F framework [20] with a focus on software design and arming novice designers with expert strategies; a programming process by Hu and co-workers [21] with a focus on generating goals and plans and converting those into a coded solution via a visual block-based programming language; and POPT [22] which has a focus on supporting software testing.

In contrast to the processes cited above, this research has a focus on all stages of problem solving when developing software solutions. This study is part of the first cycle of an action research project whose ultimate aim is the generation of an educational software development process aimed at this category of student to support their acquisition and application of problem solving skills.

3 Research Methods

The research question for this study is:

> *In the context of problem solving in software development by novice university learners, what are the subjective experiences and depth of learning of a sample cohort of freshman, university students studying software development without the support of a formal software development process?*

3.1 Participants

The control group was a cohort of first year undergraduate students who were registered on a degree in software development in the academic years 2015/16 and 2016/17. Given that the participants were not randomly assigned by the researcher, it was necessary to first conduct a pre-test to ensure they were probabilistically equivalent in order to reduce any threat to the internal validity of the experiment. This means that the confounding factor of any student having prior software development experience was eliminated. The control group had 82 students of which the gender breakdown was 70% male and 30% female. These students were tested again at the end of their second year where 16 students participated from the academic year 2016/17 and 25 students participated from the academic year 2017/18 giving a combined control group of 41 with a 75% male to 25% female gender breakdown.

3.2 Pedagogical and Assessment Process

The module that was the subject of this study was a two semester, 24 week introduction to software development which ran over the first academic year of the programme. It has

been observed in Sect. 1 of this paper that there is a gap in software engineering education in the provision of software development processes for freshman, undergraduate computing students [4]. Therefore, students in this study were taught software development in the absence of a formal software development process. This means that similar to equivalent undergraduate courses, students were primarily taught how to program in a specific language with the problem solving process to apply the language to solve problems being a suite of informal steps [6].

The programming language taught to students was Java and the order of topics taught to students are summarized in Table 1. These topics were taught via lectures and problem solving exercises given in practical sessions. Students were also taught to use pseudocode as a design technique in order to design solutions for the exercises.

Table 1. The topics taught to students (Source: Higgins and colleagues [11]).

Topics
1. Sequential Flow (e.g. using variables, display, inputs)
2. Non-sequential Flow (e.g. conditional constructs, loops)
3. Modularity (e.g. functions, parameters, scope of variables)
4. Object Oriented Interaction/Behaviour

When students were given a problem to solve, they were encouraged to analyse the problem by attempting to document on paper the requirements of the problem (i.e. a decomposition of the problem into a series of actions). Pseudocode and Java were used to design and code solutions to these requirements in an iterative and incremental cycle.

There were nine intended learning outcomes (ILOs) for this module which were used as a mechanism to test students' levels of proficiency in problem solving in software development. These ILOs are summarized in Table 2.

Table 2. Taxonomy of Intended Learning Outcomes for the module (Source: Higgins and colleagues [11]).

Taxonomy of Intended Learning Outcomes
1. Apply process of abstraction when solving problems
2. Illustrate evidence of mental modelling of programming concepts
3. Illustrate evidence of mental modelling of notional machine
4. Recognise opportunities for reuse of existing problems or sub-problems
5. Perform problem analysis and decomposition
6. Identify data that is required to solve a problem
7. Design algorithms for decomposed problems
8. Apply algorithm integration
9. Evaluate solution incrementally

3.3 Evaluation Process – AKM-SOLO

An adaption of the Kirkpatrick model was used as the evaluation process for this study. The original Kirkpatrick model is a structured mechanism with five levels which was developed as a tool for businesses to test the effectiveness of either in-house or outsourced training programmes for employees [10, 23]. However, the scope of use for this model extends beyond business as there are also many examples in the literature of the model being used to test learning interventions in an educational context for students [24, 25].

In the original Kirkpatrick model, each of the levels are deployed sequentially starting with level 1 with each subsequent level becoming increasingly complex to measure. These higher levels provide increasingly more valuable information about the overall value and impact of the training [26]. The first level – *Reaction* -measures participants' reactions to, and perceptions of, instruction received once it has been completed. The second level – *Learning* - assesses if the learning objectives for the training programme have been met. The third level - *Behaviour* – examines the behavioral change (if any) in participants as a result of instruction once they return to their jobs or future studies. The fourth level – *Results* – examines the targeted outcomes of training to an organisation such as reduced costs, improved quality and increased quantity of work. The fifth level – *Return on Investment* - measures the medium to long-term return on investment for an organisation. A return on investment is not relevant in the context of this study and in the evaluation of academic education and is therefore not considered further.

However, the Kirkpatrick model is not without its critics. Specific criticism is based on the model operating as a summative, goal-based model of evaluation with the confounding factors that can affect learning often being ignored [27]. It has also has been noted that there can be little visibility into the learning that takes place and issues that arise as a training course proceeds [28]. Furthermore, the incompleteness of the framework is troublesome; particularly the high-level nature of the levels, where there is little guidance in how to evaluate those levels [29, 30].

Therefore, for this study, the original model with its first four levels has been adapted into a model titled the Adapted Kirkpatrick Model with SOLO (AKM-SOLO). The *Structure of Observed Learning Outcomes* (SOLO) taxonomy [31, 32] is incorporated into the adaptation in order to continually monitor the depth of learning that occurs to enhance the formative nature of the model. This change makes the adapted model both a summative and formative model of evaluation. A summary description of the four levels of this model can be seen in Table 3.

Similar to the original model, this adapted model has four levels. Levels 1 and 2 are very similar to Kirkpatrick's levels 1 and 2 albeit the newly adapted level 2 evaluation is more explicit given its incorporation with SOLO. Also as can be seen from Table 3, levels 3 and 4 have been renamed in this adapted model and also given a new focus. A full description of all four levels of the model is given in the remainder of this section.

The choice of appropriate data collection instruments for this evaluation model was guided by the decision to employ a mixed methods design. Quantitative analysis was used in levels 2 and 3 of the AKM-SOLO model to evaluate a set of prescribed problems given at different stages of the academic year to test the depth of learning. Quantitative and qualitative analysis was carried out on the surveys and focus group sessions in

Table 3. A summary description of the four levels of the AKM-SOLO model framed in the context of the evaluation of a freshman, undergraduate software development course with data collection tools and mode of evaluation outlined for each level.

Level	Definition	Data collection tools	Evaluation
1. Reaction	An evaluation of students' reaction to - and experience of - the software development skills they were taught	Post-test survey and focus group	Mixed methods evaluation with triangulation
2. Learning	An evaluation of the depth of student learning that is taking place as they are being taught concepts and skills	Suite of well-defined problems across each of the four topics	SOLO Taxonomy Framework
3. Transfer	An evaluation of student software development competency at the end of first year by examining their ability to transfer their learning to solve a large, ill-defined problem	A large, ill-defined problem	SOLO Taxonomy Framework
4. Impact	An evaluation of the impact of first year problem solving in software development instruction on students' attitudes and practice in the second year of the programme	Follow-up survey	Mixed methods evaluation with triangulation

levels 1 and 4 (see Table 3). Given that this case study has a focus on understanding the learning process of freshman students studying software development for the first time, the confounding factor of prior learning is eliminated from the adapted model by subjecting students to a pre-intervention survey to ensure only novice learners are included in the evaluation.

Descriptions of the characteristics and deployment of the four levels of the model is contained in the following subsections.

Level 1 – Reaction. The aim of the first level was to document students' reaction to, and experience of, problem solving in software development. In order to achieve this aim, five research questions were posed:

1. What quantifiable engagement do students have with software development?
2. What planning techniques (i.e. analysis and design techniques) did students find useful when solving computational problems?

3. What planning techniques (i.e. analysis and design techniques) did students NOT find useful when solving computational problems?
4. Is there an association between engagement and type of technique favoured?
5. What emotional responses did students experience on the course that they perceived motivated or demotivated them in their studies?

To provide answers to these questions, students completed a survey (n = 82) and attended a focus group session (n = 21) close to the end of their first year of undergraduate study.

In an attempt to quantify students' engagement levels with problem solving, a dependent variable called *engagement* was generated from the survey. This variable had values ranging from 12, to indicate that a student is fully engaged with software development, to 0, to indicate student is not engaged. The formulation of the engagement variable involved examining 12 of the survey questions. These questions specifically examined student attitudes to the value they perceived analysis and design had when they are solving problems. Additionally, responses to these questions indicated whether the respondents would use these techniques outside of assignment work and if they planned to use them beyond the current academic year. A binary measurement score was given to the answers, which were summated to give the engagement value.

The principal quantitative techniques used on the survey data were Cronbach's alpha [33], to measure internal consistency of the data, and the Kruskal-Wallis test [34], to see if there is an association between students' level of engagement and the type of software development techniques favoured. The tool used for the quantitative analysis was IBM SPSS Version 24. The data collected from the open questions of the survey and the focus group were subjected to qualitative thematic analysis as suggested by Braun and Clark [35]. The tool used to assist in this analysis was NVivo Version 12.

Level 2 - Learning. Levels 2 and 3 of the original Kirkpatrick model required enhancement in order to have a clear and traceable process to examine learning in a formative mode. To do this, the Structure of Observed Learning Outcomes (SOLO) taxonomy [31, 32] was used to augment these levels so the depth of student learning taking place as the course progressed could be measured and issues identified. In order to test student learning in each of the four topics of interest (see Table 1), a suite of sixteen problems (four problems per topic) were given to students during the research period. As a mechanism to test the depth of student learning applied when solving these problems, a SOLO taxonomy framework was developed which mapped the five SOLO stages against the nine ILOs presented in Table 2. This framework was used as a guide by researchers to measure the depth of learning a student demonstrated in each of the nine ILOs for a specific problem (see Table 4 for a subset of this framework for illustrative purposes).

For each problem solution completed by each student (i.e. 82 students by 16 problems), the depth of learning was measured as a SOLO score for each of the nine ILOs. The SOLO score achieved was measured as a number from 1–5 to represent the SOLO stages Prestructural (1), Unistructural (2), Multistructural (3), Relational (4) and Extended Abstract (5). Calculating the mean of all nine ILO SOLO scores produced a single average SOLO score which represents the SOLO depth of learning for that problem in a

specific topic for a student. Finally, calculating the mean of all student solutions for all four problems in a topic produced a single average SOLO score for that topic.

Table 4. A subset of the SOLO Taxonomy framework as applied to the first three stages of the SOLO taxonomy in conjunction with the first three ILOs from Table 2 (Source: Higgins and colleagues [11]).

Intended Learning Outcome → / **SOLO Stage ↓**	*Applying abstraction*	*Programming Concepts*	*Notional Machine*
1: Prestructural	No understanding of abstraction	No understanding of concepts	Cannot articulate state of concept
2: Unistructural	Can abstract from problem specification to code only	Understand one of the concepts	Can articulate state of one concept
3: Multistructural	Can abstract between several levels e.g. spec – analysis, analysis – design, analysis – code, design - code but no traceability across all levels	Understand several concepts but can't relate them	Articulate states of several concepts but can't relate them

Level 3 - Transfer. Level 3 in the original Kirkpatrick model is known as *Behaviour,* as it is intended to examine employee behaviour once they return to the workplace to see how the training has impacted their work practices. In the context of this study, students are not returning to work, but an equivalent experience is *learning transfer* which is the ability of a learner to successfully apply the knowledge and skills acquired in a more realistic problem solving situation. Given that the domain of learning here is problem solving with software development, this level examines students' ability to transfer their learning from the relative containment of smaller, well-defined problems into a larger, more ill-defined problem that would mirror more closely a real-world problem. In order to better reflect this specific focus, the level is renamed *Transfer* in this adaptation. The level is evaluated using the same process as level 2 but in this case, instead of solving a suite of problems based on each of the four topics, students are presented with a large ill-defined problem to which they have to provide a solution. Solutions are measured for depth of learning in each of the nine ILOs so comparisons can be made with the results from level 2 and a conclusion reached regarding students level of proficiency in software development going forward.

Level 4 - Impact. In AKM-SOLO, the title of level 4 is renamed from Kirkpatrick's original title of *Results* to *Impact* as at this level, the focus is on examining the impact that learning how to problem solve in software development in first year has on the second year experience. The rationale for this inquiry is that it has been reported in the literature that software development habits and attitudes acquired by novice learners can be very difficult to unlearn [7]. Therefore, this level evaluates students at the end of their second year to see what positive, negative or neutral impact their first year training in problem solving has had on their attitudes to- and affective state with - software development in future years. To carry out this evaluation, a survey is given to students at the end of their second year and the evaluation is framed around the following four research questions.

1. What are students' attitudes to analysis and design in general at the end of second year?
2. What are students' attitudes to the specific analysis and design techniques they were taught in first year?
3. What recommendations do students have to improve analysis and design in first year?
4. What impact has students' first year experience had on their affective state when applying problem solving techniques to computational problems in second year?

4 Results and Findings

This section presents the results and findings from carrying out the evaluation. Full results for AKM-SOLO level 1 (research questions 1 to 4 in Sect. 3.3, Level 1 Reaction) and AKM-SOLO level 2 can be found in Higgins and colleagues [11] with a summary of those results presented in Sects. 4.1 and 4.2 of this paper. Full results from the evaluation of AKM-SOLO levels 3 and 4 can be found in Sect. 4.3 and Sect. 4.4 of this paper.

4.1 Level 1 - Reaction

This level measured students' reactions to, and experiences of, problem solving in software development where data was collected using a survey (n = 82) and running a focus group session (n = 21). A quantifiable engagement level in problem solving (see Sect. 3.3) was calculated for the cohort (n = 82) which resulted in an average score of 5.7 out of 12. This score indicates a less than average engagement with problem solving.

In examining the planning techniques (i.e. analysis and design techniques) that students find useful when solving computational problems, 42% of survey participants (n = 35) and 48% of focus group participants (n = 10) were positive about the use of analysis as a technique to help them break down the main problem into a series of ordered sub-problems which were easier to individually solve.

Examining the planning techniques that students did not find useful when solving computational problems, pseudocode was specifically cited by 46% (n = 38) of survey participants with 67% (n = 14) of focus group students indicating that they found design to be very confusing and unhelpful to them when solving computational problems.

Testing the association between the different types of analysis and design techniques favoured by students, it was seen that 78% (n = 30) of students in this study who indicated that they found no technique useful also had a very low engagement level of 0–2, with 21% (n = 8) having an engagement level of 3 and 1% (n = 1) an engagement level of 4. Conversely, 84% (n = 41) of students who indicated they favoured the technique of requirements analysis had an engagement level of 7. 48% (n = 24) of those specifically specifying pseudocode or design techniques had an engagement factor or 3 or less. This result highlights the use of pseudocode as being in negative correlation with student engagement. Conversely, of the 100% (n = 12) of students who indicated that no technique was unhelpful, 62% (n = 7) had an engagement level of 8 or more.

In examining the data from the focus group, 58% (n = 12) of students indicated that they did not carry out any design prior to attempting to code a solution and of those students, 78% (n = 9) had an engagement level of 3 or less.

The findings from the fifth research question posed in Sect. 3.3 (Level 1 – Reaction), which was outside the scope of Higgins and colleagues [11], is reported here.

What emotional responses did students experience on the course that they perceived motivated or demotivated them in their studies? In the context of the survey, when attempting to cite emotional responses that they perceived motivated them, students found this question difficult to answer as 73% (n = 60) either provided no answer or indicated that they were unsure. Of the answers that were received (n = 22), these answers are categorised and aggregated into three codes [36] – *Enjoy creating fun or useful solutions* (8.5%, n = 7)*, Enjoy writing programs* (10%, n = 8)*, Motivating to see success on course to date* (8.5%, n- = 7). When citing demotivating factors, students were much more comfortable with answering this question with 21% (n = 17) citing that there were no demotivating factors. Of the 79% (n = 64) of students who did respond, these answers were codified into *Confidence knocked from having to engage with Analysis and Design techniques (95%), Feeling bored by software development (3%), Feeling frustrated as software development too difficult (2%).*

In the context of the focus group responses, there was a large response of 81% (n = 17) of students who cited design as providing a demotivating emotional response. Students indicated that design annoyed them, that it made them lose interest in software development, that it lowered their confidence in their ability, that they hated the subject as a result, that is was a miserable experience and they would consider leaving the course as a result. For students who indicated that the design process was motivating to them, 38% (n = 8) of students responded but interestingly of that 38%, 75% (n = 6) used diagrams for design with these students indicating that switching to diagrams made them feel calmer about solving problems and gave them an overview of what they wished to achieve in their solution. This was in a context where diagrammatic techniques for design were not taught to students.

> *"Once I backed away from design using pseudocode and then started diagramming, life became much less stressful and I actually started to enjoy it"* – (Focus Group Student 07).

The remaining 2 students who submitted a positive result about design indicated that they enjoyed design when they worked with friends as it "*made me feel less alone"*

(Focus Group Student 15) and "*it's okay not to understand design initially*" (Focus Group Student 9).

4.2 Level 2 – Learning

It was observed from the findings produced at this level that the expected depth of learning for students was expected to begin at SOLO score 2.33 (just above unistructural score of 2) but it actually began at 1.99 which is just below this SOLO stage (see Fig. 1). As the students progressed through the four topics, the actual depth of learning remained lower than expected with the final question in the 4th topic producing an actual score of 3 (multistructural stage) while the expected score was 4 (relational stage).

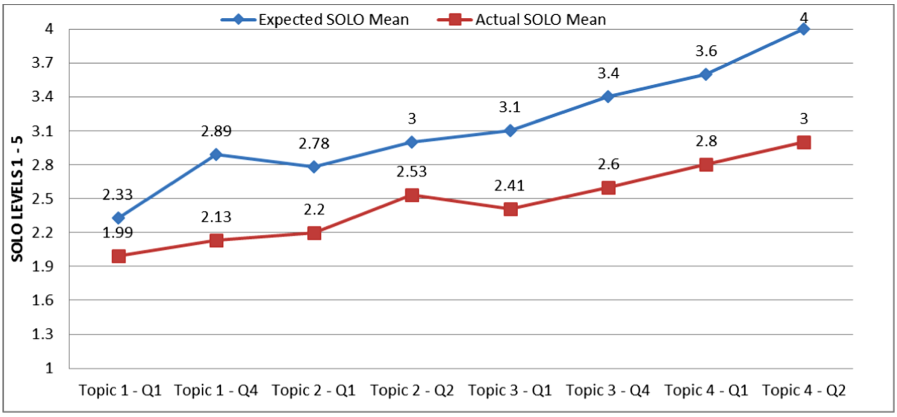

Fig. 1. Line chart to compare Expected SOLO scores with Actual SOLO scores across all four topics by students (n = 82) (Source: Higgins et al. [11]).

Therefore, even though it was expected that on average students would be able to combine multiple concepts when solving problems, in reality while they could understand and utilise several ILOs across the four topics, they had difficulties when it came to integrating ILOs to generate correct solutions. This is a low result to achieve at the end of the course as it suggests that while students can demonstrate aptitude in multiple ILOs separately, they cannot integrate them (which is the SOLO relational stage). This ability to integrate ILOs when planning and developing solutions is required if students are to become proficient problem solvers in software development. It was seen that this issue exists primarily due to students having difficulties utilising *design*, *integration* and *solution reuse* with the learning outcomes *evaluation*, *abstraction* and modelling the *notional machine* also causing significant learning issues for students. However, *understanding programming constructs, data representation* and *analysis and decomposition* were at the multistructural stage which suggests students can understand and mentally model programming concepts and variables but they find it difficult to apply that knowledge when generating solutions. A positive observation is that while the actual SOLO means for each of the four topics remained lower than the expected means, both sets of

means followed a similar upward trend meaning there was an improvement in the depth of learning.

4.3 Level 3 – Transfer

In order to test the impact of learning on students' continued ability to solve problems, a final assignment was given to students at the end of their first year, which incorporated all topics taught on the course. This was presented as a mechanism to examine students in the process of solving a larger and more ill-defined problem in comparison to the more well-defined problems that were provided for each of the four course topics (see Table 1). The students were given a basic specification that required them to research and create a retail application for a real-world organisation that sold products or services to customers. Given that students advanced to the multistructural stage (SOLO score 3) at the end of the fourth topic in level 2, it was expected that they would at least remain at this stage as they progressed through this level. Furthermore, it was expected that they would continue to improve and would move towards the relational stage (SOLO score 4) of understanding. The solutions to this problem were analysed using the SOLO taxonomy framework as introduced in Level 2 (see Sect. 3.3, Level 2- Learning). A chart presenting the percentage of students (n = 82) with their SOLO scores achieved in each of the nine ILOs is given in Fig. 2.

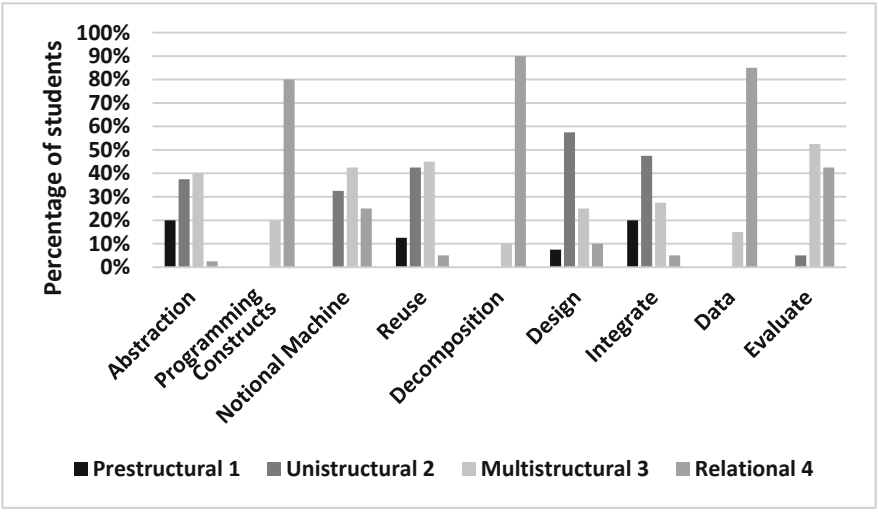

Fig. 2. AKM-SOLO Level 3 Transfer - A Clustered Column Chart showing the SOLO score percentages achieved for each of the nine ILOs from Table 2 (n = 82).

From examining the data for the intended learning outcomes (see Fig. 2), it can be seen that the ILOs related to *understanding programming constructs, analysis and decomposition, identifying data* and *evaluation* recorded a majority of students achieving a SOLO score of 4 (relational stage). This suggests that that the majority of the cohort

could successfully combine these topics when applying them to a large problem. This was an improvement from AKM-SOLO level 2 where the majority of students achieved a SOLO score of 3 (multistructural stage).

In examining the concepts of *design* and *integration,* which students found particularly difficult in level 2, there was a small improvement (see Fig. 3). For *design,* a proportion of students moved from the prestructural and unistructural depth of learning stages in level 2, resulting in small gains at the multistructural stage in level 3 (increase of 5%; n = 4) and the relational stage (increase of 8%; n = 6). In contrast, for *integration*, there was an 8% (n = 6) increase at the prestructural stage when level 3 is compared to level 2, indicating that there were more students who perceived they didn't understand the concept than was recorded in level 2. This increase in miscomprehension is not surprising given the size and complexity of integrating a solution at this level. The remaining learning outcomes of *abstraction, notional machine* and *reuse* all showed a decrease of less than 5% in understanding when moving from levels 2 to 3.

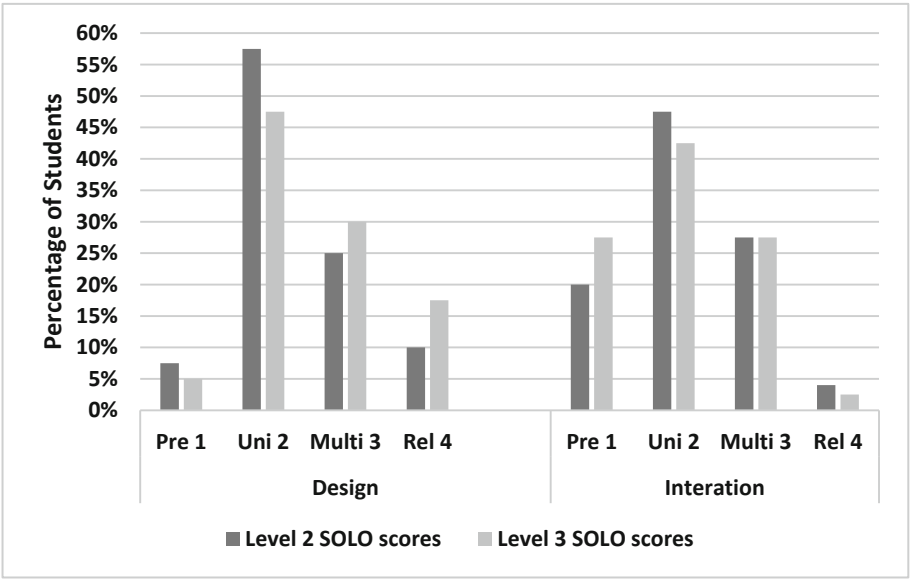

Fig. 3. Comparison of AKM-SOLO Levels 2 and 3 SOLO scores for the ILOs *design* and *integration* (n = 82).

It was seen that, on average, when all nine ILO scores are combined, 7% (n = 3) of students were still at the prestructural stage which was unchanged from level 2 and 24.7% (n = 10) were at the unistructural stage, which is an increase of 1%. However, there was a decrease of 1.7% at the multistructural stage, which is marginally mitigated by an increase of 0.6% at the relational stage, which brings that stage to 38% (n = 15). In order to measure if these percentage changes from level 2 to level 3 were statistically significant, a paired-samples t-test was conducted to compare the results between both

levels and it was found that there was no significant difference between the two levels ($t = 0$, $p > 0.05$).

4.4 Level 4 – Impact

The impact level of students' first year experience on their ongoing studies is tested at the end of their second year on the programme by participation in a survey in which 41 students participated. It should be noted that during the second year, students continued their software development education but were not explicitly taught any new analysis or design techniques. Therefore, students relied on the problem solving strategies they were taught in first year. The results from the four research questions posed (see Sect. 3.3) for this level are now presented and explored.

1. What are students' attitudes to analysis and design in general at the end of second year?

To measure attitudes to analysis and design, students were asked five closed questions which examined their approach to solving computational problems. The results from these questions are presented in Fig. 4 and Table 5. These results show a very similar pattern to the attitudes of students to analysis and design in first year, which indicate that students' attitudes have not changed in the intervening year. In the first year evaluation in Level 1, it was observed that 65% ($n = 26$) of students found analysis to be useful, whereas 46% ($n = 19$) specifically cited design (in the form of pseudocode) as not being useful, with 35% ($n = 14$) citing that neither analysis nor design were useful. A year later, where students are not explicitly taught any new analysis and design techniques but where they had more software development education, it can be seen that 56% ($n = 23$) agree that analysis is useful (see Question 1, Fig. 4). However when solving problems, 73% ($n = 31$) of students do not design solutions, but instead look for code from an apparently similar problem to modify (see Question 3, Table 5). Equally, when students are faced with logical problems in their code, 68% ($n = 28$) would try and solve the problem by continually changing their code whereas 23% ($n = 9$) would revert to design (see Question 2, Table 5). Overall, students placed little value in analysis and design with over half explicitly labeling the process as being a waste of their time when trying to solve problems (see Questions 4 and 5, Fig. 4). In contrast, 41% ($n = 17$) explicitly indicate that they see the value in analysis and design in theory which would suggest that if they were taught analysis and design as part of an integrated process with programming, there may be scope for an improvement in their engagement with planning solutions to problems.

2. What are students' attitudes to the specific analysis and design techniques they were taught in first year?

To measure students' attitudes in second year to the analysis and design techniques taught in first year, four closed questions were posed to examine how students approached solving computational problems. The results from these questions is presented in Fig. 5.

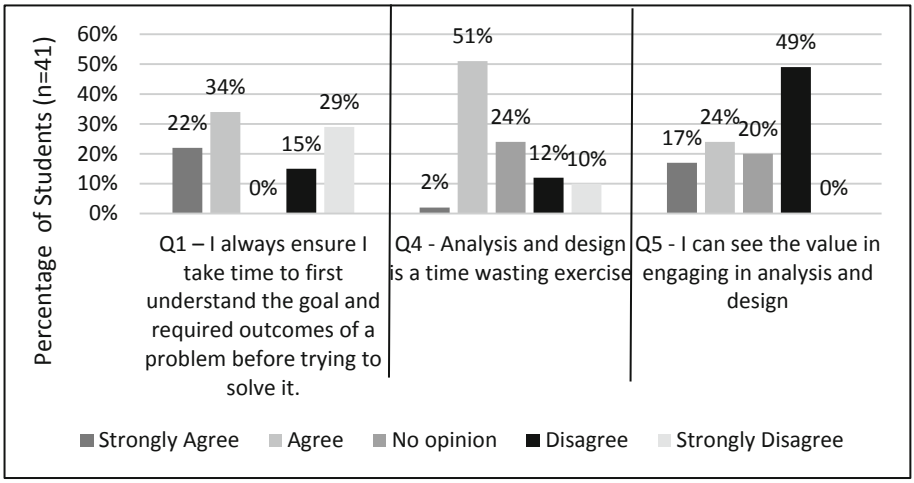

Fig. 4. AKM-SOLO Level 4 Impact - Graphed results from Likert formulated questions in examining second year students' general attitudes to analysis and design after two years of software development study (n = 41).

Table 5. AKM-SOLO Level 4 Impact - Tabular results from survey given to second years recording their attitudes to analysis and design after two years of software development study (n = 41).

Question	Result
Q2 – From the following options, indicate which option most closely matches your approach to solving logical errors in your programs	68% (n = 28) - "*keep changing the code to try and get it to work*", 17% (n = 7) "*go back to [their] design and see if [they] can find any problems in your logic*", 10% (n = 4) *"get help from a friend"*, 4% (n = 2) checked the *Other* with 2% would "*check code first and then go back to design"* 2% would *"mix between checking code and asking friends"*
Q3 – From the following options, choose the problem solving style that most closely matches your approach when solving challenging problems	73% (n = 30) - *I would try and find a similar problem* 22% (n = 9) - *I would design part of the solution first, then write code based on that design* 2.5% (n = 1) - *I would design a full solution first* 2.5% (n = 1) - *All of the above*

It can be seen that 19% (n = 8) of students perceived analysis and design to be a valuable aspect of software development; with a majority of students (63%, n = 26) stating that they found the specific analysis and design techniques taught in first year were unhelpful (see Q6 and Q7 from Fig. 5). However, this 63% was in a context where

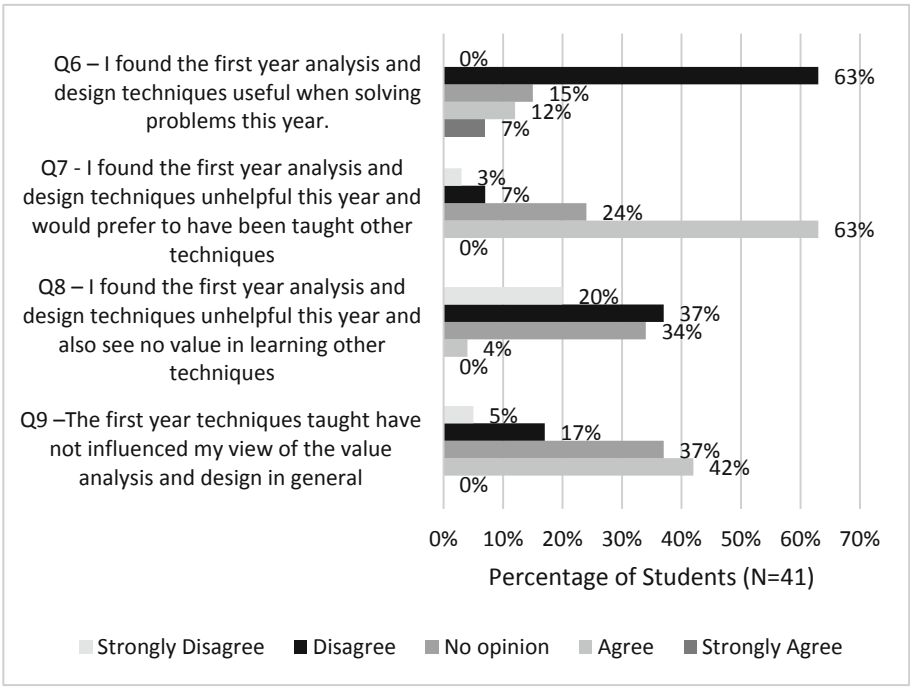

Fig. 5. AKM-SOLO Level 4 Impact - Results from survey given to second years recording their attitudes to the specific analysis and design techniques they were taught in first year (n = 41).

they did want to learn other techniques as 57% (n = 23; Q8 from Fig. 5) of students disagreed with the statement that they didn't want to learn other techniques. This suggests that students, at least theoretically, see the value in analysis and design and are open to learning other planning strategies which is also supported by 42% (n = 17; Q9 from Fig. 5) of students indicating that the choice of techniques taught has not influenced their opinion of analysis and design in general. Unsurprisingly, of the students who gave examples of impediments to learning in first year, 62% (n = 12) specifically cited pseudocode as being unhelpful to them in first year with 21% (n = 4) indicating they saw no value to analysis and design in general.

3. What recommendations do students have to improve analysis and design in first year?

Almost half (44%; n = 19) of students answered this question; and the responses were categorised, by the researcher, into three themes with many students suggesting more than one theme. 59% (n = 24) suggested diagrammatic techniques should be used for analysis and design; 32% (n = 13) indicated that analysis and design should be explicitly included in their second year of study and finally, 73% (n = 30) of students wanted classes dedicated to providing a definite strategy for how to solve problems.

4. What impact has students' first year experience had on their affective state when applying problem solving techniques to computational problems in second year?

To examine the impact of first year on students' affective levels, students were asked three closed questions. The results from these questions is presented in Table 6.

Table 6. AKM-SOLO Level 4 Impact - Results from survey given to second years to examine their emotional responses to analysis and design in second year (n = 41).

Question	Result
Q13 – Indicate the impact of your first year experience learning how to solve problems on your motivation to solve problems in second year as positive, negative or neutral	Positive – 33% (n = 10) Negative - 47% (n = 14) Neutral – 20% (n = 6)
Q14 – Rate on a 5 point Likert scale your current confidence level in software development from none to very confident	Very Confident – 7% (n = 3) Reasonable Confidence – 32% (n = 13) Low Confidence – 44% (n = 18) No Confidence – 17% (n = 7)
Q15 - Rate on a Likert scale your level of interest in working as a developer following graduation with points from *Definitely* to *Definitely Not*	Definitely future career – 29% (n = 12) Probably future career – 27% (n = 11) Don't know – 22% (n = 9) Probably not future career – 22% (n = 9) Definitely not future career – 0%

Almost half of students (47%, n = 14, Q13 from Table 6) indicated that their first year experience of learning how to analyse and design solutions to problems had a negative impact on their motivation to plan solutions to problems in second year. This lack of planning is also borne out in students' confidence levels, with 61% (n = 25, Q4 from Table 6) indicating they had low or no confidence in their ability to solve software problems. This negative affective impact is also reflected in 44% (n = 18, Q5 from Table 6) of students who either don't know or definitely feel that this is not a future career for them. This means that by the end of second year, students are negatively affected by analysis and design which is impacting both their confidence and motivation in their software development studies.

5 Discussion

From the application of the AKM-SOLO model in this study, it has been observed that students' overall attitudes to - and affective states when - problem solving in software development is not encouraging. From the findings in the last section, it can be observed that students regard engaging in software development to be primarily about programming with the concept of designing solutions in particular considered not to be useful and is avoided where possible. It was also seen that this attitude carries through to the end of their second year on the programme. This is not an unexpected result given that it

has been cited in the literature that getting students to design solutions rather than try to program a solution through trial and error or memorizing other solutions is very difficult [20, 37]. Nonetheless, this is a worrying result especially as the issue is not that students do not have the aptitude to be software developers, but rather that they are not developing the analysis and design skills that allow them engage them with the problem solving nature of the discipline as a whole. As educators, we wish them to become developers who can design and implement solutions but they are inadvertently being taught with a focus on being programmers instead and this is affecting their proficiency and positive attitude to software development.

Student engagement is generally considered to be a predictor of learning [38]. However, it has been noted that computer science students' general level of engagement in their studies has been recorded internationally as being much lower than students from other disciplines [39]. Therefore, the relatively low engagement level of 5.7 out of 12 found in this study is not surprising as it suggests that a majority of students are not adequately engaged with the topic and that is borne out in the consistently underperforming set of actual SOLO scores acquired across the four topics taught in first year. Interestingly, 94% (n = 82) of the first year survey respondents view the process of programming as being more important than the analysis and design stages which suggests that they don't see the value in carrying out planning prior to writing a program. This is an issue also observed by Garner [40] and it has been found that this lack of focus on planning is a lead issue in the development of maladaptive cognitive practices [7]. The results from this study suggest that student engagement in the process of solving software development problems is directly aligned to how useful they find the process of carrying out analysis and design. If the process of analysis and design wasn't objectively important in software development, then students would be able to skip this stage and move directly to coding, and their engagement level would not be affected which has not been observed here. This is not a new observation as the importance of structuring problem solving into analysis and design strategies for novices has been recognised in the literature [41, 42]. Therefore, as the engagement level is low and their depth of learning in analysis and design is not at a SOLO relational stage, this suggests that if students can't successfully participate in analysis and design, this affects their ability to engage fully with their studies to become proficient developers.

On examining the findings, most students found the process of analysis (i.e. breaking a problem into a series of sub-tasks that need to be solved) to be a useful activity to help them start solving a problem. This is typical top-down analysis which has long been proven as a mechanism to support students [43]. This is reflected both in the responses from students in the focus group and survey as well as the improvement seen in SOLO scores for the ILO *analysis and decomposition* across the four topics. However, despite this positive experience, this ILO is still not at the SOLO relational stage that would be expected of students at the end of their first year, which suggests further structure in carrying out analysis would help. Students need to be able to visualise and create mental models in order to understand "what" needs to be done to solve a problem. However, it has been observed that most students find such mental modelling difficult [44]. Therefore, adding a visualisation technique to the analysis process could be useful

in helping students both carry out analysis as well as engage in the mental modelling required.

The area of design is a seriously divisive issue for students. It has been found in other studies that design is typically a much harder task for novice learners than programming due to; the need for complex mental modelling of computing constructs to take place in order to design a solution and also the issues with understanding pseudocode and its inherent lack of feedback [40, 45, 46]. Likkanen & Perttula [47] also observe that even if students successfully complete design in a top-down fashion where they decompose a problem into sub-problems, they often then experience difficulties in integrating the sub-problem solutions back into a final solution. These issues with design are also reflected in this study where it is very clear that pseudocode as a design technique is not fit for purpose; most students find it neither useful nor helpful. From the survey findings in research question 3 in Sect. 4.1, it can be seen that novice learners find it difficult to understand the role of pseudocode as a mechanism to abstract from the technicalities of a programming language and instead see it as yet another language they have to learn. This language issue with pseudocode was observed by Hundhausen et al. [48]. Students also criticized the lack of support and structure in this design technique which they find makes it difficult to use effectively. This difficulty is reflected by many students indicating that they move immediately to the coding phase before they have adequately decomposed a problem or carried out at least some design for a solution. From the focus group findings, this issue also emerges where it can also be seen that this issue with pseudocode is biasing students against their perception of design as being a useful process.

This difficulty with design is also reflected in the SOLO scores where the ILOs of *design*, *integration* and *solution reuse* were found to have the lowest SOLO scores across the four topics; signaling students have a specific issue with these topics. Equally the ILOs involving the *mental modelling of the notional machine*, the use of *abstraction* and the *evaluation of solutions* also returned consistently low scores.

As an alternative to pseudocode, it was seen from the survey findings in Sect. 4.1 that some students successfully gravitated towards using design techniques such as flowcharts to support them in designing algorithms despite it not being taught. Given that flowcharts have been cited in the literature as a very credible mechanism for visualising a flow of control in an algorithm [49] and that they also are a natural visualisation technique, such charts could be a very useful alternative to help students engage in the process of design.

These negative attitudes to design were borne out when examining the results obtained when students were asked to develop a solution to an ill-defined, larger problem at the end of their first year. While students demonstrated marginal improvements in design, they were in general ill-equipped to analyse and design a solution and instead reverted to surface strategies of trying to program a solution without a plan. Students particularly exhibited problems with integrating incremental solutions into a final solution as can be seen in Fig. 2. Given these issues students have with design and integration in solving a larger problem, it is not surprising then to observe that students also performed poorly in being able to model the notional machine, carry out reuse and utilise abstraction to enable them plan a solution at different levels of detail. This is a worrying result, as going forward in their studies, students will naturally be expected to be able

to solve more complex and ill-defined problems which require the active planning and modelling of solutions.

In their second year, students received a full year of tuition in advanced programming which would have improved their technical knowledge. However, as was seen in the results for AKM-SOLO level 3, this had no impact on the negative regard they had for the problem planning techniques taught in first year which is carried through their second year. 66% (n = 27) of second year students indicated they did not find their first year experience in problem planning to be useful and an equal number of students specifically cited pseudocode as being an impediment to learning. This result is also borne out in the literature as Hu [50] in synthesising research from [9, 51–53] found that increased educational attainment in software development has little effect on students' valuation of design as a useful process. However, encouragingly 57% (n = 23) disagreed with the statement that they would not like to learn other planning techniques which would tentatively suggest that they do see the value in planning even theoretically. From an affective perspective, 47% (n = 19) of second year students indicated that their first year experience had a negative impact on their motivation to solve problems in second year with 61% (n = 25) of students indicating they have poor or low confidence levels in software development. Such results are a concern as they suggest a high proportion of students are at risk of either leaving or failing to proceed in the programme. Overall, these results highlight the important role that first year analysis and design has in forming effective software development habits that will enable students to grown their proficiency and affective state as they proceed through their studies. This view is also supported by Hu [50] who argues for the use of an explicit design process when teaching software development as opposed to the global norm of using informal design strategies.

In summary, the results produced less than satisfactory findings around the issue of problem solving for software development coupled with a low level of engagement. Therefore, it can be concluded that if students perceive they are not appropriately supported in the development process by the use of appropriate development techniques, this has a negative impact on their engagement levels with software development. This impact can negatively affect their chances of continuing, and succeeding, in their course as well as deciding to pursue a career in software development. These findings suggest that in order for students to engage in problem solving in software development that they need to be properly scaffolded and supported by a software development process to guide them in acquiring good development planning habits as they set out on their learning journey. This suggestion for an explicit process is explicitly backed up by 73% (n = 30) of second year students who indicated they wanted classes dedicated to providing a definite strategy for how to solve problems.

6 Conclusions

There has been over thirty years of research into researching and proposing new pedagogical approaches to teaching software development to freshman, undergraduate students. However, despite the valuable innovations that this research has produced, there are still ongoing issues recorded globally with proficiency and retention in comparison with other undergraduate programmes. This case study examined the learning experiences of

an undergraduate first year cohort who were studying software development as novices. It was found that the absence of a formal software development process for this cohort resulted in students attempting to program solutions to problems with little interest or engagement in problem planning and this issue continued into their second year of undergraduate study. In general, students could not see the benefit in carrying out analysis and design for problem solving and this not only affected their proficiency in software development but also had a negative impact on their desire to work in the software industry. These results suggest that the provision of an educational software development process, aimed specifically at first year novice learners, could have a positive impact on their learning and attitudes to problem solving in software development.

References

1. United States Department of Labor.: Computer and Information Technology Occupations. https://www.bls.gov/ooh/computer-and-information-technology/home.htm. Assessed 2 Feb 2018
2. Stachel, J., Marghitu, D., Brahim, T.B., Sims, R., Reynolds, L., Czelusniak, V.: Managing cognitive load in introductory programming courses: A cognitive aware scaffolding tool. J. Integr. Des. Process Sci. **17**(1), 37–54 (2013)
3. Whalley, J., Kasto, N.: A qualitative think-aloud study of novice programmers' code writing strategies. In: Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education, pp. 279–284. ACM, Uppsala (2014)
4. Caspersen, M.E., Kolling, M.: STREAM: a first programming process. ACM Trans. Comput. Educ. **9**(1), 1–29 (2009)
5. Suo, X.: Toward more effective strategies in teaching programming for novice students. In: IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE), pp. T2A-1--T2A-3 (2012)
6. Coffey, J.W.: Relationship between design and programming skills in an advanced computer programming class. J. Comput. Sci. Coll. **30**(5), 39–45 (2015)
7. Huang, T.-C., Shu, Y., Chen, C.-C., Chen, M.-Y.: The development of an innovative programming teaching framework for modifying students' maladaptive learning pattern. Int. J. Inf. Educ. Technol. **3**(6), 591–596 (2013)
8. Simon et al.: Predictors of success in a first programming course. In: Proceedings of the 8th Australasian Conference on Computing Education, vol. 52, pp. 189–196. Australian Computer Society, Inc., Hobart (2006)
9. Loftus, C., Thomas, L., Zander, C.: Can graduating students design: revisited. In: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education, pp. 105–110. ACM, Dallas (2011)
10. Kirkpatrick, D.L.: Education Training Programs: The Four Levels, 3rd edn. Berrett-Kohler, San Francisco (1994)
11. Higgins, C., O'Leary, C., McAvinia, C., Ryan, B.: A study of first year undergraduate computing students' experience of learning software development in the absence of a software development process. In: Lane, H., Zvacek, S., Uhomoibhi, J. (eds.) CSEDU 2019–11th International Conference on Computer Supported Education, 2019. SCITEPRESS, Heraklion, Crete (2019)
12. Pears, A., et al.: A survey of literature on the teaching of introductory programming. ACM SIGCSE Bull. **39**(2), 2004–2023 (2007)

13. Guo, P.J.: Online python tutor: embeddable web-based program visualization for CS education. In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education, pp. 579–584. ACM, Denver (2013)
14. Gautier, M., Wrobel-Dautcourt, B.: artEoz-dynamic program visualization. In: International Conference on Informatics in Schools, pp. 70–71, Münster, Germany (2016)
15. Mozelius, P., Shabalina, O., Malliarakis, C., Tomos, F., Miller, C., Turner, D.: Let the students contruct their own fun and knowledge-learning to program by building computer games. In: European Conference on Games Based Learning, pp. 418–426. Academic Conferences International Limited, Porto, Portugal (2013)
16. Trevathan, M., Peters, M., Willis, J., Sansing, L.: Serious games classroom implementation: teacher perspectives and student learning outcomes. In: Society for Information Technology & Teacher Education International Conference, pp. 624–631. Association for the Advancement of Computing in Education (AACE), Savannah, Georgia (2016)
17. Dahiya, D.: Teaching software engineering: a practical approach. ACM SIGSOFT Softw. Eng. Notes **35**(2), 1–5 (2010)
18. Savi, R., von Wangenheim, C.G., Borgatto, A.F.: A model for the evaluation of educational games for teaching software engineering. In: 25th Brazilian Symposium on Software Engineering (SBES), pp. 194–203. IEEE, Sao Paulo (2011)
19. Rodriguez, G., Soria, Á., Campo, M.: Virtual Scrum: a teaching aid to introduce undergraduate software engineering students to scrum. Comput. Appl. Eng. Educ. **23**(1), 147–156 (2015)
20. Wright, D.R. Inoculating novice software designers with expert design strategies. In: American Society for Engineering Education. ASEE (2012)
21. Hu, M., Winikoff, M., Cranefield, S.: A process for novice programming using goals and plans. In: Proceedings of the Fifteenth Australasian Computing Education Conference, vol. 136, pp. 3–12. Australian Computer Society, Inc, Adelaide (2013)
22. Neto, V.L., Coelho, R., Leite, L., Guerrero, D.S., Mendon, A.P.: POPT: a problem-oriented programming and testing approach for novice students. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 1099–1108. IEEE Press, San Francisco (2013)
23. Kirkpatrick, D.: Kirkpatrick, Evaluating Training Programs: The four levels, 3rd edn. Berrett-Koehler Publications, San Francisco (2013)
24. Chang, N., Chen, L.: Evaluating the learning effectiveness of an online information literacy class based on the Kirkpatrick framework. **64**(3), 211–223 (2014)
25. Byrne, J.R., Fisher, L., Tangney, B.: A 21st century teaching and learning approach to computer science education: teacher reactions. In: Zvacek, S., Restivo, M.T., Uhomoibhi, J., Helfert, M. (eds.) CSEDU 2015. CCIS, vol. 583, pp. 523–540. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29585-5_30
26. Reio, T.G., et al.: A critique of kirkpatrick's evaluation model. New Horizons Adult Educ. Hum. Resource Dev. **29**(2), 35–53 (2017)
27. Guerci, M., Bartezzaghi, E., Solari, L.: Training evaluation in Italian corporate universities: a stakeholder-based analysis. Int. J. Training Dev. **14**(4), 291–308 (2010)
28. Hayes, H., et al.: A formative multi-method approach to evaluating training. Eval. Program Planning **58**, 199–207 (2016)
29. Aluko, F.R., Shonubi, O.K.: Going beyond Kirkpatrick's Training Evaluation Model: the role of workplace factors in distance learning transfer. Africa Educ. Rev. **11**(4), 638–657 (2014)
30. Alliger, G.M., et al.: A meta-analysis of the relations among training criteria. Pers. Psychol. **50**(2), 341–358 (1997)
31. Biggs, J.B., Collis, K.F.: Evaluation the Quality of Learning: the SOLO Taxonomy (Structure of the Observed Learning Outcome). Academic Press (1982)
32. Biggs, J.B., Collis, K.F.: Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome). Academic Press (2014)

33. Cronbach, L.J.: Coefficient alpha and the internal structure of tests. Psychometrika **16**(3), 297–334 (1951)
34. Kruskal, W.H., Wallis, W.A.: Use of ranks in one-criterion variance analysis. J. Am. Stat. Assoc. **47**(260), 583–621 (1952)
35. Braun, V., Clarke, V.: Using thematic analysis in psychology. Qual. Res. Psychol. **3**(2), 77–101 (2006)
36. Beins, B.C., McCarthy, M.A.: Research Methods and Statistics. Cambridge University Press (2017)
37. Garner, S.: A quantitative study of a software tool that supports a part-complete solution method on learning outcomes. J. Inf. Technol. Educ. (2009)
38. Carini, R.M., Kuh, G.D., Klein, S.P.: Student engagement and student learning: Testing the linkages. Res. High. Educ. **47**(1), 1–32 (2006)
39. Sinclair, J., et al.: Measures of student engagement in computer science. In: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education. ACM (2015)
40. Garner, S.: A program design tool to help novices learn programming. In: ICT: Providing Choices for Learners and Learning (2007)
41. Deek, F., Kimmel, H., McHugh, J.A.: Pedagogical changes in the delivery of the first-course in computer science: problem solving, then programming. J. Eng. Educ. **87**(3), 313–320 (1998)
42. Morgado, C., Barbosa, F.: A structured approach to problem solving in CS1. In: Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education. ACM, Haifa (2012)
43. Ginat, D., Menashe, E.: SOLO taxonomy for assessing novices' algorithmic design. In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education. ACM (2015)
44. Cabo, C.: Quantifying student progress through Bloom's taxonomy cognitive categories in computer programming courses. In: ASEE Annual Conference and Exposition, Conference Proceedings (2015)
45. Lahtinen, E., Ala-Mutka, K., Järvinen, H.-M.: A study of the difficulties of novice programmers. In: ACM SIGCSE Bulletin. ACM (2005)
46. Hummel, H.G.K.: Feedback model to support designers of blended learning courses. Int. Rev. Res. Open Distrib. Learn. **7**(3) (2006)
47. Liikkanen, L.A., Perttula, M.: Exploring problem decomposition in conceptual design among novice designers. Des. Stud. **30**(1), 38–59 (2009)
48. Hundhausen, C.D., Brown, J.L.: What You See Is What You Code: A "live" algorithm development and visualization environment for novice learners. J. Vis. Lang. Comput. **18**(1), 22–47 (2007)
49. Paschali, M.E., et al.: Tool-assisted Game Scenario Representation Through Flow Charts. In: ENASE (2018)
50. Hu, C.: Can students design software?: The answer is more complex than you think. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education. ACM, Memphis (2016)
51. Eckerdal, A., et al.: Can graduating students design software systems? ACM SIGCSE Bull. **38**(1), 403–407 (2006)
52. Eckerdal, A., et al.: Categorizing student software designs: methods, results, and implications. Comput. Sci. Educ. **16**(3), 197–209 (2006)
53. Tenenberg, J.D., et al.: Students designing software: a multi-national, multi-institutional study. Inf. Educ. **4**(1), 143–162 (2005)