2019-05-01

A Study of First Year Undergraduate Computing Students' Experience of Learning Software Development in the Absence of a Software Development Process

Catherine Higgins
Technological University Dublin, catherine.higgins@tudublin.ie

Claire McAvinia
Technological University Dublin, claire.mcavinia@tudublin.ie

Ciaran O'Leary
Technological University Dublin, ciaran.oleary@tudublin.ie

Barry J. Ryan
Technological University Dublin, barry.ryan@tudublin.ie

Recommended Citation

Higgins, C. et al. (2019) A STUDY OF FIRST YEAR UNDERGRADUATE COMPUTING STUDENTS' EXPERIENCE OF LEARNING SOFTWARE DEVELOPMENT IN THE ABSENCE OF A SOFTWARE DEVELOPMENT PROCESS, *11th International Conference on Computer Supported Education* CSEDU 19 , Heraklion, Crete 2nd – 4th May 2019.

A STUDY OF FIRST YEAR UNDERGRADUATE COMPUTING STUDENTS' EXPERIENCE OF LEARNING SOFTWARE DEVELOPMENT IN THE ABSENCE OF A SOFTWARE DEVELOPMENT PROCESS

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

Technological University Dublin, Ireland

Abstract

Despite the ever-growing demand for software development graduates, it is recognised that a significant barrier for increasing graduate numbers lies in the inherent difficulty in learning how to develop software. 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. As a means of understanding how such learners problem solve in software development in the absence of a formal process, this study examines the experiences and depth of learning acquired by a sample set of novice, freshman university learners. 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.

Keywords: Software Development Education, Freshman University Students, Software Development Process

1. Context for Study

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 (2015) 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 (Stachel et al., 2013). 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 (Whalley and Kasto, 2014). This suggests 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 (Caspersen and Kolling, 2009). 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 (Coffey, 2015; Suo, 2012). 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

Presented at the 11th International Conference on Computer Supported Education – CSEDU 19 – Heraklion, Crete 2nd – 4th May 2019.

www.csedu.org

analysis or design and coding by rote learning (Huang et al., 2013). These practices can be very difficult to unlearn and can ultimately prohibit student progression in the acquisition of software development skills (Huang et al., 2013; Simon et al., 2006).. It has also been found that problems in designing software solutions can persist even past graduation (Loftus et al, 2011).

To address these challenges, this paper describes a focussed 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. In this focussed study, the experiences and depth of learning of a sample set of novice, freshman university learners being taught software development in the absence of a formal software development process is reported. The aim of the study is to identify specific issues and behaviour that can arise in the absence of such a process.

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 (Pears et al., 2007). 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 (Gautier and Wrobel-Dautcourt, 2016; Guo, 2013); and the use of game based learning as a basis for learning programming and game construction (Mozelius et al., 2013; Trevathan et al., 2016).

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 (2010) who presents a study of teaching software engineering using an applied approach to postgraduate and undergraduates with development experience, Savi and co-workers (2011) who describe a model to assess the use of gaming as a mechanism to teach software engineering and Rodriguez (2015) 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 (Caspersen and Kolling, 2009) which focus on design in an object oriented environment; the P^3F framework (Wright, 2012) with a focus on software design and arming novice designers with expert strategies; a programming process by Hu and co-workers (2013) with a focus on generating goals and plans and converting those into a coded solution via a visual block-based programming language; and POPT (Neto et al., 2013) 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 were 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.

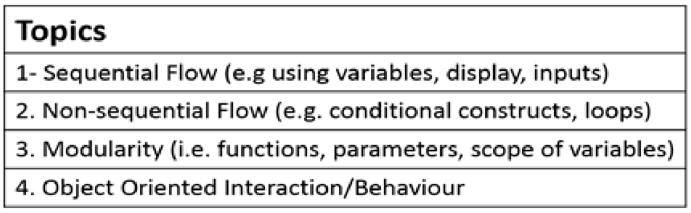
3.2 Pedagogical and Assessment Process

The module that was the subject of this study was a two semester, 24 week introduction to software development which ran over the entire first academic year of the programme. It has been observed in Section 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 (Caspersen and Kolling, 2009). 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 (Coffey, 2015).

The programming language taught to students was Java and the order of programming 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 to the exercises.

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). Then they were taught to use pseudocode to design and illustrate the principal computational constructs needed to address these requirements. Finally, the designed requirements were tested by converting the pseudocode into Java and integrating the tested code into the finalised program.

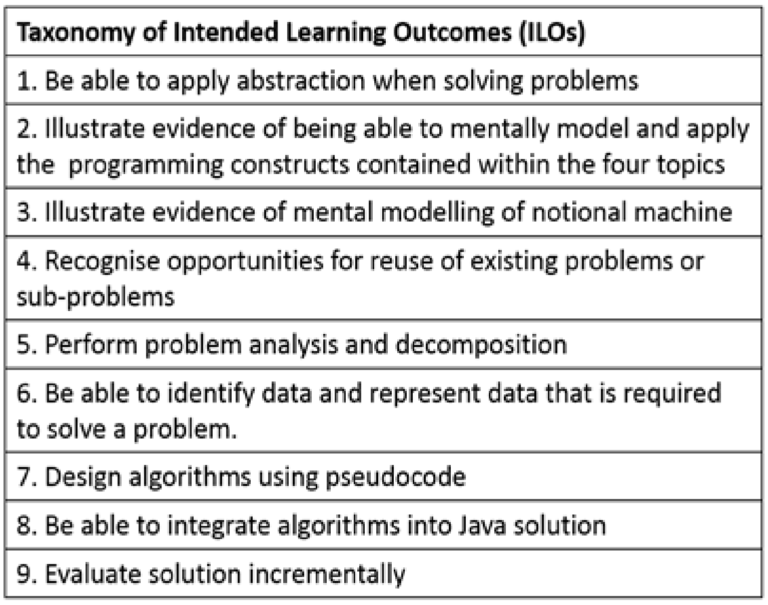
Table 1: The topics taught to the participants



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

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

Table 2: Taxonomy of Intended Learning Outcomes for participants.



Taxonomy of Intended Learning Outcomes (ILOs)
1. Be able to apply abstraction when solving problems
2. Illustrate evidence of being able to mentally model and apply the programming constructs contained within the four topics
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. Be able to identify data and represent data that is required to solve a problem.
7. Design algorithms using pseudocode
8. Be able to integrate algorithms into Java solution
9. Evaluate solution incrementally

3.3 Data Collection and Evaluation Methodology

In deciding on appropriate data collection instruments for this study, this choice was guided by the decision to employ a mixed methods design. Quantitative analysis was used 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 data collected from an end-of year survey and focus group session to ascertain students' reactions to - and experiences of - that learning.

In structuring the evaluation of the data, the Kirkpatrick framework was used. This framework is a structured mechanism with five levels which can be used by businesses to test the effectiveness of either in-house or out-sourced training programmes for employees (Kirkpatrick, 1994). There are also many examples in the literature of this framework being used to test learning interventions for students (Byrne et al, 2015; Chang and Chen, 2014).

For the purposes of this paper, which is examining students' experiences and depth of learning, only a subset of this framework is presented. This subset contains level 1 which focuses on student reaction to learning and level 2 which focuses the depth of learning acquired. When working with this framework, it was the contention of this researcher that level 2 required adaption in order to have a clear and traceable process to examine learning. To do this, the Structure of Observed Learning Outcomes (SOLO) taxonomy (Biggs and Collis, 1982, 2014) was used to augment the abridged Kirkpatrick framework in order to measure the depth of student learning that has taken place.

3.3.1 Measuring Level 1 - Reaction to, and Experience of, Problem Solving in Software Development

In measuring students' reaction to, and experience of, problem solving in software development, four 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?

To provide answers to these questions, participants completed a survey (n=82) and attended a focus group session (n=21).

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 down 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 perceive analysis and design has when they are solving problems as well as an indication of whether they would use these techniques outside of assignment work and if they plan 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.

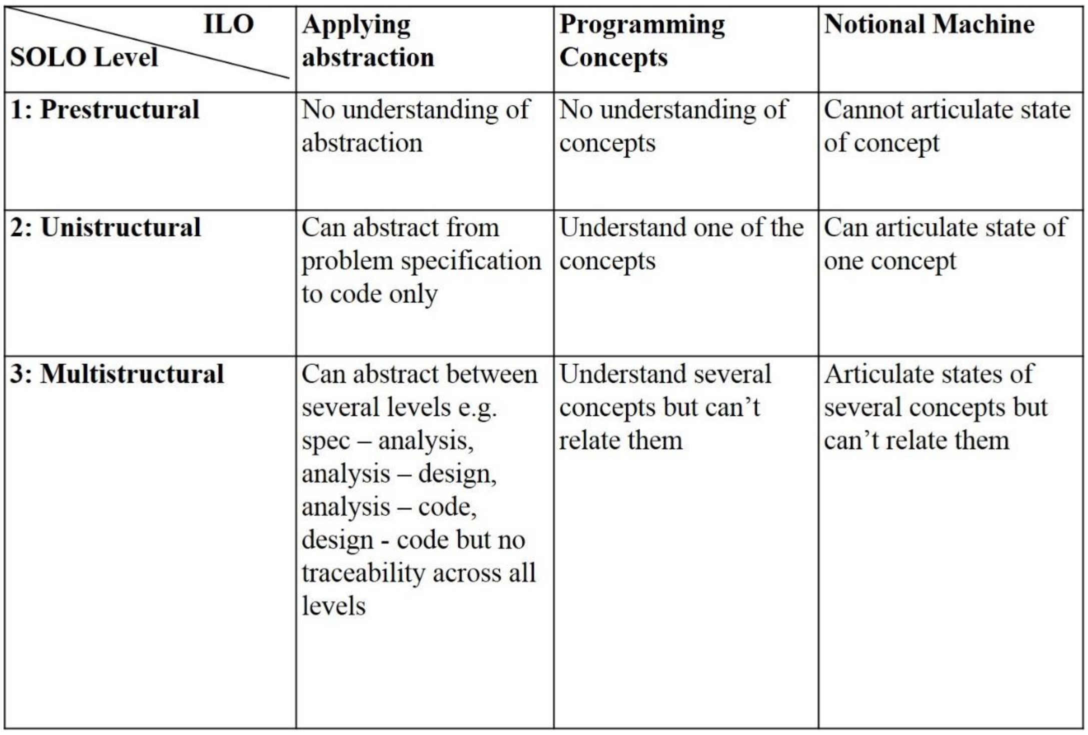
The principal quantitative techniques used on the survey data were Cronbach's alpha (1951) to measure internal consistency of the data and the Kruskal-Wallis test (1952) to see if there is an association between students' level of engagement and the type of software development techniques favoured. The 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 (2006). The tool used to assist in this analysis was NVivo Version 12.

3.3.2 Measuring Level 2 - Depth of Learning

In order to test student learning in each of the four topics summarised in Table 1, a suite of sixteen problems (four problems per topic) was given to students during the year. 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 levels 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. A subset of this framework is given in Table 3 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 level score for each of the nine ILOs. The SOLO level achieved was measured as a number from 1 – 5 to represent the SOLO levels 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 a SOLO level 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 level score for that topic.

Table 3: A subset of the SOLO Taxonomy framework as applied to the first three levels of the SOLO taxonomy in conjunction with the first three ILOs from Table 2.



ILO **SOLO Level**	**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

Prior to giving the problems to the students, each problem was examined by the researcher and a peer reviewer with the aim of approximating the least set of SOLO scores that would be expected from students. To test the reliability between researcher and peer reviewer scores, a kappa coefficient value of 0.7 was generated which is deemed acceptable as a reliability test (Viera and Garrett, 2005). A similar mode of using SOLO to estimate what is expected from students is presented by many researchers in this space (Brabrand and Dahl, 2009; Izu et al, 2016; Sheard et al., 2008; Shuhidan et al, 2009).

4. Results and Findings

This section presents the results and findings from carrying out this study.

4.1 Level 1 - Reaction to, and Experience of, Problem Solving in Software Development

1. What quantifiable engagement do students have with software development?

The engagement level (see Section 3.3.1) was calculated for each student (n=82) and this resulted in an average score of 5.7 out of 12. 68% (n=56) of the cohort scored between 3 and 8 with 70% (n=39) of that group scoring between 3 and 6 inclusively with the other 30% scoring between 6 and 8.

2. What planning techniques (i.e. analysis and design techniques) did students find useful when solving computational problems?

Results from quantitative analysis on the survey are illustrated in Figure 1.

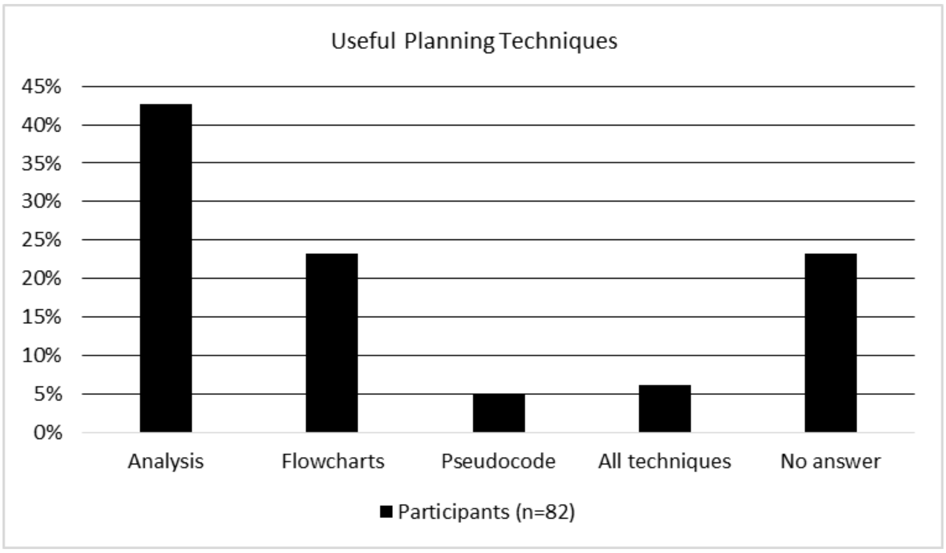


Figure 1: Categories and percentages of planning techniques that students found useful (n=82).

In examining the focus group and open questions of the survey, 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.

"*The lecturer gives you a big problem, doesn't it make sense to break into smaller problems so now you have maybe 4 smaller and easy to understand problems than one big one that I haven't a notion about?*"- (Focus group Student 03)

On the question of the usefulness of design, only three students in the focus group spoke positively about the usefulness of design.

3. What planning techniques (i.e. analysis and design techniques) did students NOT find useful when solving computational problems?

Results from quantitative analysis on the survey are illustrated in Figure 2 where pseudocode was specifically cited by 46% of students as not useful. When asked for reasons for this finding (n=38), the answers are categorised into three themes *Pseudocode is another language so why not just use Java* (47%, n=18), *Don't know where to start as confusing to use* (26%, n=10), N*o feedback from pseudocode so can't tell if it's right or wrong* (26%, n=10).

In examining the data from the focus group, 67% (n=14) indicated that they found design to be very confusing and unhelpful to them in problem solving.

I don't know how to start with this design technique or how to use it to help me think about solutions. It doesn't help me only stresses me out more as I'm confused all the time. – (Focus Group Student 21).

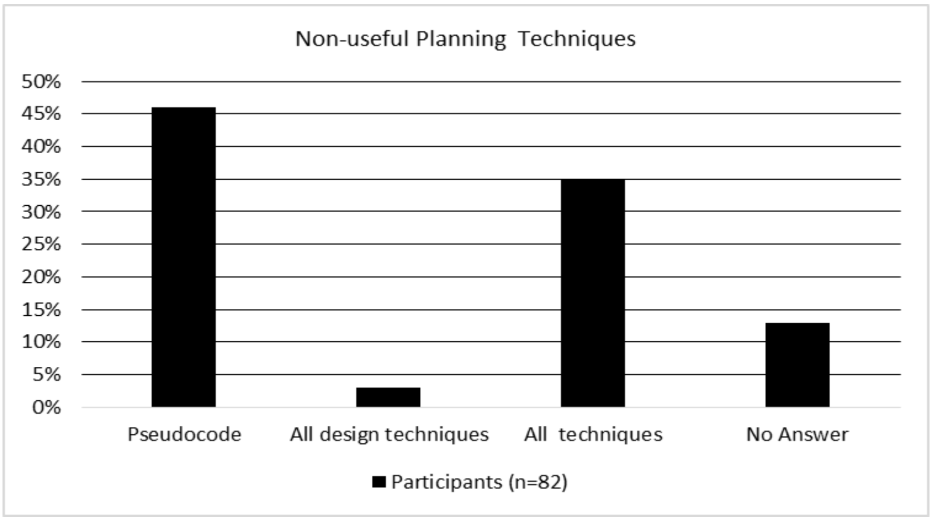


Figure 2: Categories and percentages of planning techniques that students did not find useful (n=82).

In examining how useful or not they found analysis and design in general, students from the survey were also asked if they engaged in analysis and design when solving complex problems; and in a separate question, they were also asked if they felt coding was more important than analysis and design. Only 11% (n=9) indicated that they would engage with analysis and design when solving complex problems with 94% (n=77) agreeing with the statement that coding was more important than planning.

4. Is there an association between engagement and type of technique favoured?

In testing the association between the different types of analysis and design techniques favoured by students, the evaluation carried out by a Kruskal-Wallis test showed that there is a statistically significant difference in engagement levels between the different types of techniques favoured by students ($p<0.05$, Kruskal-Wallis, Chi-Square=60.4, df=6, N=82).

Examining this further, 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.

Additionally, in testing the association between the different types of analysis and design techniques not favoured by students, the evaluation indicated that there is a statistically significant difference in engagement levels between the different types of techniques not favoured by students ($p<0.05$, Kruskal-Wallis, Chi-Square=74.23, df=4, N=82). Examining this further, it was seen that 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.

I should say that I do all the planning stuff but that would be a lie! I look up programs you've [lecturer] given us based on the topics that the assignment is based on and see if I can use those to try and put together a solution - (Focus Group Student 19)

I just reverse engineer my code and I don't mind saying that out loud as it's what you've [lecturer] said on all of my feedback. I can't write pseudocode so the only thing to do is try and figure it out in java and then go back and turn that into pseudocode but even that doesn't work as it's obvious to you what I did so it's useless – (Focus Group Student 04)

On the other hand, even though only 19% (n=4) of students indicated that planning a solution via analysis and design is important, all of these students achieved an engagement level of 7 or more. This aligns with the findings in the quantitative analysis that an engagement with analysis and design has a direct impact on student's overall engagement levels with software development as a whole.

4.2 Level 2 Depth of Learning

This section presents the findings from analysing and assessing the student answers to the problems devised for the four topics as outlined in Table 1.

Figure 3 summarises these findings as a line chart to show the expected and actual SOLO levels achieved. The SOLO level scoring for question 1 (Q1) of a topic indicates the SOLO level achieved at the start of the topic with the SOLO level scoring for question 4 (Q4) indicating the level achieved towards the end of the topic.

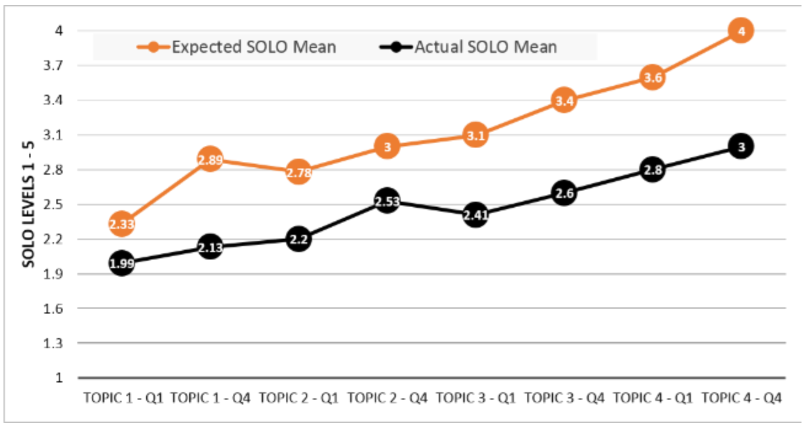


Figure 3: Line chart to compare Expected SOLO scores with Actual SOLO scores across all four topics by students (n=82)

4.2.1 Topic 1 – Sequential Flow Control

This topic ran from weeks 1 to 4 of the academic year and focussed on simple foundational aspects such as defining primitive variables, assigning and inputting values into variables, updating variable values and displaying variable values. By the end of week 2, question 1 was given to students with an expected SOLO score of 2.33 which means that on average students are expected to be past the unistructural stage (which has a score of 2) where they can understand and apply one ILO and are moving towards the multistructural stage (score 3) where they can utilise more than one ILO. At the end of topic 1 when they are given question 4, the expected score is 2.89 which means that on average students are expected to be almost at the multistructural stage where they can comfortably apply and utilise more than one ILO in the context of this topic when problem solving. In contrast, the actual scores for questions 1 (given in week 2) and 4 (given in week 4) are 1.99 and 2.13 respectively which means they are on average on the unistructural level where they can utilise just one ILO in the context of this topic. Drilling into these results found that students began the topic with specific issues with the ILOs associated with *design*, *integration* and the *notional machine* which were all at the prestructural level. By the end of the topic, on average students moved into the multistructural level of understanding for the ILOs related to *understanding program concepts* and *data representation;* with *abstraction* and the *notional machine* at the unistructural level and the remaining ILOs at the prestructural level.

4.2.2 Topic 2 - Non-Sequential Flow Control

This topic ran from weeks 5 to 9 and focussed on conditional and iterative constructs. Both constructs involve the testing of conditions which - based on the truth of the condition - will either selectively chose which path of the solution to execute (conditional constructs) or will repeatedly execute part of a solution until the condition is false (iterative construct). For questions 1 (given in week 5) and 4 (given in week 9), the expected SOLO scores were 2.78 and 3.0 in contrast to the actual SOLO scores of 2.2 (for question 1) and 2.55 (for question 4) respectively. Drilling into these scores found that on average students were still at the unistructural level, however they were moving towards the multistructural level. The ILOs that showed the most improvement across this topic were those relating to *analysis* (decomposition), *understanding programming constructs* and *data representation*.

4.2.3 Topic 3 – Modularity

This topic ran from weeks 10 to 15 and focussed on the integration of sequential /conditional / iterative constructs into a subprogram that carries out one defined action. Such a subprogram may or may not return a value and may take input parameters. For questions 1 (given in week 10) and 4 (given in week 15), the expected SOLO scores were 3.1 to 3.4 in contrast to the actual SOLO scores of 2.41 (for question 1) to 2.6 (for question 4) respectively. Drilling into these scores found that the topic of modularity is difficult in general for students due to its use of local, global variables and parameter passing (Park et al., 2015), hence the dip in actual scores from the last topic. The percentage of students still at the prestructural level for all ILOs at this stage was 17% (n=14). The ILOs of *abstraction*, *notional machine*, *design, evaluation* and *integration* were still at unistructural stage with no students at the multistructural level in *abstraction*. The *solution reuse* ILO was still at the prestructural level in general and it is both this outcome as well as *abstraction* that kept that high number of students at the prestructural level. *Understanding programming constructs, data representation* and *analysis* were at the multistructural level.

4.2.4 Topic 4 - Object Oriented Interaction/Behaviour

This topic ran from weeks 16 to 24 and focused on the definition of new data types in the form of classes where the new data types have a range of data values and a set of defined actions. For questions 1 (given in week 17) and 4 (given in week 24), the expected SOLO scores were 3.6 to 4.0 in contrast to the actual SOLO scores of 2.8 (for question 1) to 3 (for question 4) respectively. Drilling into these scores, it was found that at the end of the course, students had barely achieved the multistructural level of learning. It can be seen that this level of learning exists primarily due to issues with *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* were at the multistructural level which suggests students can understand and mentally model programming concepts and variables but they find it difficult to apply that knowledge when solving problems.

4.2.5 Summary of SOLO findings

Overall, it can be seen from the findings in this section and the measurements summarized in Figure 3 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. In the observed actual SOLO means, students began with an average score of 1.99 which is just on the cusp of the unistructural level of learning and they finished with a score of 3.0 which indicates they moved to the multistructural stage of learning. This means that on average, students could understand and utilise several ILOs across the four topics but they had difficulties when it came to integrating ILOs to improve problem solving.

This is a low result to achieve at the end of the course as it suggests that while students can demonstrate multiple ILOs separately, they cannot integrate them (which is the SOLO relational level). This ability to integrate ILOs when planning and developing solutions is required if students are to become proficient problem solvers in software development.

5. Discussion

Student engagement is generally considered to be a predictor of learning (Carini, Kuh, and Klein, 2006). 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 (Sinclair et al., 2015). 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. Interestingly, 94% (n=82) of the 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 (2007) and it has been found that this lack of focus on planning is a lead issue in the development of maladaptive cognitive practices (Huang et al., 2013). 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. Also in support of this observation is the fact that the importance of structuring problem solving into analysis and design strategies for novices has been recognised for many years (Deek et al, 1998; Morgado and Barbosa, 2012). Therefore, as the engagement level is low and their depth of learning in analysis and design is not at a SOLO relational level, 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 (Ginat and Menashe, 2015). This is reflected both in the responses from students in the focus group and survey as well as the improvement seen in SOLO levels for the ILO *Problem Analysis and Decomposition* across the four topics. However, despite this positive experience, this ILO is still not at the SOLO relational level 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 (Cabo, 2015). 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, the issues with understanding pseudocode and its inherent lack of feedback (Garner, 2007; Lahtinen et al,2005). Likkanen & Perttula (2009) 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 Section 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 (2007). 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 level scores where the ILOs of *design*, *integration* and *solution reuse* were found to have the lowest SOLO scores across the four topics; signalling 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 Section 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 (Paschali et al., 2018) 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.

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.

Overall, 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.

6. Conclusions

Finding new and improved methods of teaching software development to freshman, undergraduate students is an extensively researched area, but with little consensus. In this study, it was found that the provision of an appropriate software development process for this cohort is an area requiring more focus and structure. As a first step in the development of such a process, this study examined the experience and depth of learning acquired by undergraduate, novice software development students during their first year of study in the absence of a formal software development process. The findings from this study suggest that without an appropriate level of scaffolding, especially in analysis and design, students' attitudes and proficiency in developing software solutions can be compromised as they rush to try and implement solutions without appropriate planning. The next stage of this research project is in the generation and implementation of a software development process to provide this scaffolding.

References

Biggs, J.B., Collis, K F. 1982. Evaluation the Quality of Learning: The Solo Taxonomy (Structure of the Observed Learning Outcome), Academic Press.

Biggs, J.B., Collis, KF. 2014. Evaluating the Quality of Learning: The Solo Taxonomy (Structure of the Observed Learning Outcome). Academic Press.

Brabrand, C and Dahl, B (2009) Using the Solo Taxonomy to Analyze Competence Progression of University Science Curricula. Higher Education, 58(4), 531-49.

Braun, V and Clarke, V (2006) Using Thematic Analysis in Psychology. Qualitative research in psychology, 3(2), 77-101.

Byrne, J R, Fisher, L and Tangney, B. (2015). A 21st Century Teaching and Learning Approach to Computer Science Education: Teacher Reactions, International Conference on Computer Supported Education (pp. 523-40): Springer.

Cabo, C (2015) Quantifying Student Progress through Bloom's Taxonomy Cognitive Categories in Computer Programming Courses.

Carini, R M, Kuh, G D and Klein, S P (2006) Student Engagement and Student Learning: Testing the Linkages. Research in higher education, 47(1), 1-32.

Caspersen, M E and Kolling, M (2009) Stream: A First Programming Process. Trans. Comput. Educ., 9(1), 1-29.

Chang, N and Chen, L (2014) Evaluating the Learning Effectiveness of an Online Information Literacy Class Based on the Kirkpatrick Framework. Libri, 64(3), 211-23.

Coffey, J W (2015) Relationship between Design and Programming Skills in an Advanced Computer Programming Class. J. Comput. Sci. Coll., 30(5), 39-45.

Cronbach, L J (1951) Coefficient Alpha and the Internal Structure of Tests. psychometrika, 16(3), 297-334.

Dahiya, D (2010) Teaching Software Engineering: A Practical Approach. ACM SIGSOFT Software Engineering Notes, 35(2), 1-5.

Deek, F, Kimmel, H and McHugh, J A (1998) Pedagogical Changes in the Delivery of the First-Course in Computer Science: Problem Solving, Then Programming. Journal of Engineering Education, 87(3), 313-20.

Garner, S (2007) A Program Design Tool to Help Novices Learn Programming. ICT: Providing choices for learners and learning.

Gautier, M and Wrobel-Dautcourt, B (2016) Arteoz-Dynamic Program Visualization. ISSEP 2016, 70.

Ginat, D and Menashe, E. (2015). Solo Taxonomy for Assessing Novices' Algorithmic Design, Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 452-7): ACM.

Guo, P J. (2013). Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education, Proceeding of the 44th ACM technical symposium on Computer science education (pp. 579-84): ACM.

Hu, M, Winikoff, M and Cranefield, S. (2013). A Process for Novice Programming Using Goals and Plans, Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136. Adelaide, Australia: Australian Computer Society, Inc.

Huang, T-C, Shu, Y, Chen, C-C and Chen, M-Y (2013) The Development of an Innovative Programming Teaching Framework for Modifying Students' Maladaptive Learning Pattern. International Journal of Information and Education Technology, 3(6), 591.

Hundhausen, C. D., & Brown, J. L. (2007). What You See Is What You Code: A “live” algorithm development and visualization environment for novice learners. Journal of Visual Languages & Computing, 18(1), 22-47.

Izu, C, Weerasinghe, A and Pope, C. (2016). A Study of Code Design Skills in Novice Programmers Using the Solo Taxonomy, Proceedings of the 2016 ACM Conference on International Computing Education Research (pp. 251-9): ACM.

Kirkpatrick, D L. (1994). Education Training Programs: The Four Levels: San Francisco: Berrett-Kohler.

Kruskal, W H and Wallis, W A (1952) Use of Ranks in One-Criterion Variance Analysis. Journal of the American statistical Association, 47(260), 583-621.

Lahtinen, E, Ala-Mutka, K and Järvinen, H-M. (2005). A Study of the Difficulties of Novice Programmers, ACM SIGCSE Bulletin (Vol. 37, pp. 14-8): ACM.

Liikkanen, L A and Perttula, M (2009) Exploring Problem Decomposition in Conceptual Design among Novice Designers. Design studies, 30(1), 38-59.

Loftus, C, Thomas, L and Zander, C. (2011). Can Graduating Students Design: Revisited, Proceedings of the 42nd ACM technical symposium on Computer science education. Dallas, TX, USA: ACM.

Morgado, C and Barbosa, F. (2012). A Structured Approach to Problem Solving in Cs1, Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. Haifa, Israel: ACM.

Mozelius, P, Shabalina, O, Malliarakis, C, Tomos, F, Miller, C and Turner, D. (2013). Let the Students Contruct Their Own Fun and Knowledge-Learning to Program by Building Computer Games, European Conference on Games Based Learning (pp. 418): Academic Conferences International Limited.

Neto, V L, Coelho, R, Leite, L, Guerrero, D S and Mendon, A P. (2013). Popt: A Problem-Oriented Programming and Testing Approach for Novice Students, Proceedings of the 2013 International Conference on Software Engineering. San Francisco, CA, USA: IEEE Press.

Park, J, Esmaeilzadeh, H, Zhang, X, Naik, M and Harris, W. (2015). Flexjava: Language Support for Safe and Modular Approximate Programming, Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. Bergamo, Italy: ACM.

Paschali, M E, Bafatakis, N, Ampatzoglou, A, Chatzigeorgiou, A and Stamelos, I. (2018). Tool-Assisted Game Scenario Representation through Flow Charts, ENASE (pp. 223-32).

Pears, A, Seidman, S, Malmi, L, Mannila, L, Adams, E, Bennedsen, J, Devlin, M and Paterson, J (2007) A Survey of Literature on the Teaching of Introductory Programming. ACM SIGCSE Bulletin, 39(2), 19.

Rodriguez, G, Soria, Á and Campo, M (2015) Virtual Scrum: A Teaching Aid to Introduce Undergraduate Software Engineering Students to Scrum. Computer Applications in Engineering Education, 23(1), 147-56.

Savi, R, von Wangenheim, C G and Borgatto, A F. (2011). A Model for the Evaluation of Educational Games for Teaching Software Engineering, Software Engineering (SBES), 2011 25th Brazilian Symposium on (pp. 194-203): IEEE.

Sheard, J, Carbone, A, Lister, R, Simon, B, Thompson, E and Whalley, J L (2008) Going Solo to Assess Novice Programmers. SIGCSE Bull., 40(3), 209-13.

Shuhidan, S, Hamilton, M and D'Souza, D. (2009). A Taxonomic Study of Novice Programming Summative Assessment, Proceedings of the Eleventh Australasian Conference on Computing Education-Volume 95 (pp. 147-56): Australian Computer Society, Inc.

Sinclair, J, Butler, M, Morgan, M and Kalvala, S. (2015). Measures of Student Engagement in Computer Science, Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (pp. 242-7): ACM.

Simon, Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., Raadt, M.D., Haden, P., Hamer, J., Hamilton, M., Lister, R., Petre, M., Sutton, K., Tolhurst, D., Tutty, J., 2006. Predictors of Success in a First Programming Course. Proceedings of the 8th Australasian Conference on Computing Education - Volume 52. 189-196. Australian Computer Society, Inc.

Stachel, J, Marghitu, D, Brahim, T B, Sims, R, Reynolds, L and Czelusniak, V (2013) Managing Cognitive Load in Introductory Programming Courses: A Cognitive Aware Scaffolding Tool. Journal of Integrated Design and Process Science, 17(1), 37-54.

Suo, X (2012) Toward More Effective Strategies in Teaching Programming for Novice Students. Teaching, Assessment and Learning for Engineering (TALE), 2012 IEEE International Conference on, T2A-1-T2A-3.

Trevathan, M, Peters, M, Willis, J and Sansing, L. (2016). Serious Games Classroom Implementation: Teacher Perspectives and Student Learning Outcomes, Society for Information Technology & Teacher Education International Conference (Vol. 2016, pp. 624-31).

United States Department of Labor. (2015). Computer and Information Technology Occupations. Retrieved August 17, 2018, https://www.bls.gov/ooh/computer-and-information-technology/home.htm

Viera, A J and Garrett, J M (2005) Understanding Interobserver Agreement: The Kappa Statistic. Fam Med, 37(5), 360-3.

Whalley, J and Kasto, N. (2014). A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies, Proceedings of the 2014 conference on Innovation & technology in computer science education. Uppsala, Sweden: ACM.

Wright, D R. (2012). Inoculating Novice Software Designers with Expert Design Strategies, American Society for Engineering Education: American Society for Engineering Education.