A Software Development Process for Freshman Undergraduate Students

Catherine Higgins, Fredrick Mtenzi, Ciaran O'leary, Orla Hanratty, Claire Mcavinia

A Software Development Process for Freshman Undergraduate Students

Catherine Higgins[1], Fredrick Mtenzi[1], Ciaran O'Leary[1], Orla Hanratty[1] and Claire McAvinia[1]

[1]Dublin Institute of Technology, Aungier St, Dublin 2, Ireland
{catherine.higgins, fredrick.mtenzi, ciaran.oleary, orla.hanratty, claire.mcavinia}@dit.ie

Abstract. This conceptual paper presents work which is part of an ongoing research project into the design of a software development process aimed at freshman, undergraduate computing students. The process of how to plan and develop a solution is a topic that is addressed very lightly in many freshman, undergraduate courses which can leave novices open to developing habit-forming, maladaptive cognitive practices. The conceptual software development process described in this paper has a learning process at its core which centres on declarative knowledge (in the form of threshold concepts) and procedural knowledge (in the form of computational thinking skills) scaffolding freshman software development from initial planning through to final solution. The process - known as Computational Analysis and Design Engineered Thinking (CADET) - aims to support the structured development of both software and student self-efficacy.

Keywords. Introductory software development process · computational thinking · threshold concepts

1. Introduction

A software development process is a mechanism which informs a software developer of the steps and stages involved in developing quality software from initial analysis to final design and implementation [1]. Even though there are many software development processes available for experienced developers, very little work has been carried out on developing appropriate processes for freshman, 3rd level learners [2]. This lack of appropriate software development processes presents a vacuum for educators which means that software analysis and design is typically taught very informally and implicitly on introductory courses at 3rd level with an emphasis instead on teaching a programming language [3-6]. Unless they are guided to do otherwise, novices will often jump straight into implementing some aspect of a solution without any planning because they can find it difficult to separate ideas for solutions from the implementation of those ideas [7, 8]. This can lead to novices adopting maladaptive cognitive practices in software development, particularly surface practices (e.g. coding by rote learning) which can be very difficult to unlearn and can ultimately prohibit student progression in the acquisition of software development skills [9]. It has also been found that problems in designing software solutions can persist even to

graduation [10]. Therefore, it follows that if a software development process is incorporated explicitly in an appropriate way into introductory courses to scaffold students in software development, this could limit the development of such maladaptive practices.

This paper describes a conceptual and dynamic software development process which has been devised for undergraduate freshman learners. Section 2 describes related research while section 3 gives a short overview of the framework on which the process is based. Section 4 describes the factors that guided the operationalision of the framework into a software development process. Section 5 describes the process and section 6 concludes the paper with a discussion of the contribution this paper makes to software engineering educational research.

2. Related research

There has been a wealth of research over many decades into software development education within the context of improving retention and development proficiency at 3rd level. Research has focused on many areas such as reviewing the choice of programming languages and paradigms suitable for novice learners with a wide variety of languages suggested from commercial, textual languages through to visual block-based languages [11]; the development of visualisation tools to create a diagrammatic overview of the notional machine as a user traces through programs and algorithms [12, 13]; and the use of game based learning as a basis for learning programming and game construction [14, 15].

Research that specifically looks at software development processes for introductory courses at 3rd level have a tendency to focus attention on a particular stage of the development process. Examples are the STREAM process [2] which focuses on design in an object oriented environment; the P^3F framework [16] with a focus on software design and arming novice designers with expert strategies; a programming process by Hu et al [17] which focuses on generating goals and plans and converting those into a coded solution via a visual block-based programming language; POPT [18] which has a focus on supporting software testing; and Morgado & Barbosa's process [19] which aims to support students from problem presentation to the development of a solution though the use of template forms coupled with an instructor supplied prototype. The process described in this paper is similar to Morgado & Barboso's process in that it aims to support all stages of developing software but the focus here is based on the provision of a process that can grow with students' experience. The process is not tied to any particular programming paradigm but its use is assumed to be in the context of imperative, commercial programming languages which are commonly taught at 3rd level [20].

3. Computational Analysis and Design Engineered Thinking (CADET) Framework

Prior to the development of a software development process, it was important to formulate a framework on which the process will be based. The role of this

framework is to guide the context and content of the resulting software development process. The first issue that required attention was in understanding the context in which the software development process would be used. This is an environment where freshman undergraduate students typically have little or no programming experience and are learning how to develop software solutions in a systematic fashion. This brought up an interesting question – should students be taught how to program first and then be introduced to a software development process or should programming concepts and skills be taught as part of a process? This research takes the latter view as teaching students how to program independently of process runs the risk of students developing poor development habits that become ingrained by the time they learn a process. Therefore, the software development process is scaffolded so that it inherently encompasses a learning process which can slowly fade as students gain expertise of developmental concepts, practices and grow their self-efficacy. The relationship between learning process and software development process is visualised in figure 1 where the 4 stages of competence model [21] is used to timeline the progression of learning.

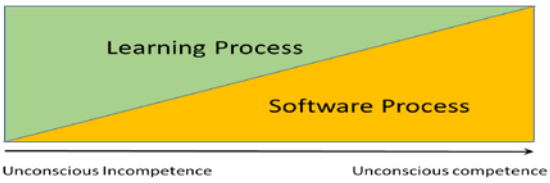

Fig. 1. From Learning Process to Software Development Process (Source: Author)

Initially, the learner is categorised as an unconscious incompetent who doesn't know what they need to know so the software development process is heavily scaffolded as a learning process where students are guided to use the software development process to solve a suite of problems that are appropriate to each stage of their learning. By the time the user has gained experience of the foundational developmental concepts and practices, the scaffolding of the learning process will be removed to allow the learner continue to use the software development process in solving new and more complex problems as they expand their learning and continue their journey towards becoming unconscious competents.

Once the context of the environment was understood, a conceptual framework was devised and developed in order to fully identify the components and activities in the learning process. The full details of the background, rationale for - and development of - the framework can be found in reference [22]. A diagrammatic overview of the framework is given in figure 2.

Concepts (threshold stages)	Practices (CT skills)	Perspectives (affective issues)
TC1. State and Sequential Flow TC2. Non-Sequential Flow Control TC3. Modularity TC4. Object Behaviour	CT1. Abstraction CT2. Data Representation CT3. Decomposition CT4. Evaluation of solutions (Testing, Debugging, Critiquing) CT5. Pattern recognition CT6. Generating Algorithms (modelling and simulation)	AI1. Self-Efficacy

Fig. 2. The CADET Framework (Source: Author)

In summary, the **concepts** represent the declarative knowledge that students need in order to be able to understand and use programming constructs. These concepts are categorised as four threshold concepts stages [18]. *TC1 State and Sequential Flow* involves gaining an understanding of "simple" data items (e.g. characters, numbers and strings) and how their state changes when sequential actions are carried out on them. *TC2 Non-sequential Flow Control* keeps the focus on state but adds complexity to this idea by presenting more complex actions such as iteration and how these actions affect state and flow control. *TC3 Modularity* introduces modularity and how that affects state and especially flow control. Finally, *TC4 Object Behaviour* - which is optional and is only used in an object-oriented environment - examines the idea of objects and the connection between state and behaviour and how objects interact and activate each other's behaviour.

The **practices** represent the procedural knowledge that students need in order to be able to apply the above concepts when solving problems. These practices are categorised as computational thinking skills and are codified as skills CT1 – CT6 in column 2 of figure 2. Finally, the **perspectives** are the affective issues that impact learning which are considered to be embodied in self-efficacy.

This framework marries current research into threshold concepts, computational thinking and affective learning to produce a framework that supports declarative knowledge (threshold concepts), procedural knowledge (computational thinking) and affective learning issues [18]. Learning these knowledge areas is facilitated by instruction and by repeatedly solving problems using Pólya's problem solving model [23] which has been adapted to suit the context of this research [18]. The framework (and subsequent process) is known as computational analysis and design engineered thinking (CADET).

4. Operationalisation of Framework to Process

As part of the operationalisation and development of the framework into a software development process, current best practice in both the teaching of software development and in software development processes for professional developers is considered for inclusion into the process.

4.1 Best Practice in Teaching Software Development

There are two basic approaches to teaching software development – top-down and bottom up. The top-down stepwise refinement approach originated in the 1970s by Wirth [24] and involves breaking down a problem into a series of levels with tasks. One advantage of the top-down approach is that a high-level overview of the solution is first constructed which can then be slowly broken down into its constituent parts. However, critics of top-down design state that it involves creating a monolithic design where coding cannot begin until the design is fully complete [25]. The bottom-up approach starts from a finely granulated specification of the problem which is generated by identifying and implementing the smallest tasks. These tasks are then combined to form larger tasks with this successive amalgamation of smaller tasks into larger tasks continuing until the entire solution is implemented. A very high level view of the solution is not available at the start of the process which can prove

problematic for novices who typically find it difficult to reassemble tasks back into a full solution [26].

In comparing expert developers to novices, experts have a breadth first, top down approach to formulating solutions whereas novices tend to have a depth first, bottom up approach where they focus on specific aspects of the problem [26, 27]. However, as noted above, novices can then find it difficult to re-integrate the different parts of the problem into a final solution and may revert to trial and error approaches to find something that works [26]. On the other hand, experts use strategies based on their experience to avoid trial and error [16] which suggests that novices need to be supplied with scaffolded strategies to help them problem solve as they gain experience.

This research suggests a hybrid approach - between top down and bottom up development - as an attempt to keep novices focused on the big picture while allowing them to use a depth first approach. This approach has been coined by this researcher as a "*design down, code up*" approach where solutions are visually designed by students in a scaffolded, top down fashion; code is produced for low level designs which gives feedback to the students who are then supported in combining these tasks to effectively code up to a final solution.

In the context of applying an appropriate learning theory, research into computer science education has several successes using constructivist and constructionist theory [28-30]. Social constructivism occurs when learning is perceived as an active process and where individual knowledge is constructed through solving problems in a collaborative exercise. This theory forms the basis of the development process described in this paper as the students will carry out extensive problem solving to construct their own individual knowledge and will engage in Vygotsky's theory of the "more able other" [31] by participating in paired development and in articulating solutions to the class cohort. Therefore, the learning process for this software development process has been designed with the aim of facilitating constructivist learning.

4.2 Best Practice in Software Development Processes

As well as ensuring that best practice in the teaching of software development is incorporated into the software development process described in this paper, it is also important to consider and include current best practice in existing software development processes. One way of incorporating best practice is to align this process with the philosophy of verifiably successful software development processes. Given that most modern software development projects use Agile processes [32], this is the category of process chosen to represent best practice. Kastl et al [33] has demonstrated how the philosophy and general characteristics of Agile processes can be adapted as a guide for best practice. This means that the core characteristics that govern all Agile processes will be used to guide the operation of this process. These characteristics include the use of iterative and incremental development, adaptive modelling, refactoring of development artefacts and paired programming.

5. Computational Analysis and Design Engineered Thinking (CADET) Software Development Process

The software development process operates as a 4 stage problem solving model based on an adapted version of Pólya's model as described in the CADET framework [22]. The four stages of the model are *1. Understand the problem, 2. Break into tasks, 3. Design and Code, 4. Evaluate solution and learning.* During the learning process stage, learners will work in pairs and will be taught the threshold concept stages which make up the declarative knowledge. This learning aspect of the software development process is represented as a ladder of learning where each concept is ordered and is a prerequisite to learning the next concept. Each concept is taught via instruction and the computational thinking skills required to ustilise the concept are acquired by solving a suite of problems using the 4 stage adapted problem solving model which is supported by an Agile philosophy. Each stage of the problem solving model will use a subset of computational thinking skills. The process is summerised in figure 3.

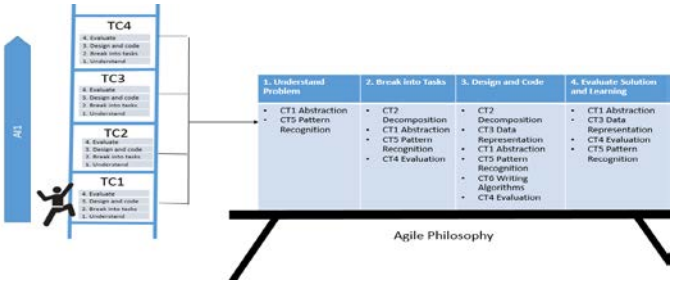

Fig. 3. CADET Software Development Process (Source: Author)

When all 4 threshold concept stages have been taught and practiced, students will continue to use the 4 stage problem solving model with associated computational thinking practices as the basis for the software development process. The software development process is augmented by a support tool which will provide a platform to provide learners with problems to solve as well as diagrammatic tools to support their analysis, design and reflective work. While it is expected that student's self-efficacy will grow and wane as they attempt to solve problems, it is hoped that the scaffolded environment based on social constructivist learning will allow the student's self-efficacy to generally grow in tandem with their knowledge (identified as A1 in the vertical arrow beside the ladder of learning in figure 3). This will be measured by student reflection. Each of the 4 stages of the problem solving model are now described in more detail.

1. *Understand the problem* - Using the support tool, learners will be invited to articulate their understanding of either a problem that they have provided or a problem that is provided to them as part of the learning process stage. This articulation of understanding is achieved by employing the computational thinking skills of *functional abstraction* to generate a high-level summary of the problem and *pattern recognition* to see if the problem is similar to any previous problems

that the learner may have solved. This high level summary is recorded in the support tool.

2. *Break into tasks* - This stage employs *decomposition* to convert the high-level summary and specification from stage 1 into an intermediate set of constituent tasks and to further refine those tasks into more basic tasks if required. In order to make this stage visual, the tool supports students brainstorming candidate tasks using a mind map where their problem summary is the central task. Mind mapping has been shown to be successful in helping learners to brainstorm and specifically in analysing software solutions [34]. The map will be refined into ordered tasks and subtasks. The support tool will facilitate leaners to utilise *abstraction* to visually trace backwards and forwards from the high-level summary from stage 1 into this stage to ensure consistency between the stages. *Pattern recognition* will be employed by learners to identify any tasks that have been used in previous problems and colour coding will be employed to identify any complex tasks that need to be designed.
3. *Design and Code* - This stage employs *decomposition* to take a task and generate an algorithm represented as a flow chart (or optionally a class diagram if operating in an object oriented paradigm) for the task. This stage also involves *data representation* and *algorithm writing* to represent the computational steps needed to represent a task solution as a flowchart with a level of detail to make it easy for the task to be converted into program code. All tasks will be designed, coded and evaluated in an iterative manner until correct and then reintegrated into a growing final product. The support tool will facilitate leaners to visually utilise *abstraction* to oscillate between tasks identified in the mind map and any associated designs and code to ensure consistent mapping between stages.
4. *Evaluate Solution and Learning* - This stage allows learners to reflect on their solution from start to finish and employ abstraction to zoom in and out of the solution to understand it at the various functional and data abstraction levels. The support tool will prompt learners to employ critiquing mechanisms to see if any aspect of the solution could have benefited from using analysis, design or coding artefacts from previous problems or if the solution can be optimized by identifying any duplication. Learners will be required to reflect on and articulate their learning.

When the process is being employed solely as a software development process, learners will be able to use both the process and associated support tool by providing their own specification for a problem and working through each of the above stages to systematically develop their final solution.

6. Discussion

Despite the acknowledged importance of using software development processes both in the software industry and in education, this research has identified a gap in software engineering education in the provision of appropriate software development processes for freshman, undergraduate computing students in a context where learners predominately have no prior programming experience. One reason for this gap is due

to the problematic nature of teaching software processes to novices. A software development process gives guidance to developers in the development of software solutions from analysis through to final product but for commercial processes, it is assumed that the developer has pre-existing programming knowledge. This makes the use of such processes difficult for educators of introductory software development courses and produces a conundrum in how to support students in the use of development processes in the absence of programming knowledge. In such an environment, it is natural that the focus of such courses will gravitate towards the teaching of programming concepts first with the topic of development process coming later in the course or in later years. The problem with such a strategy is that it allows students to potentially develop maladaptive cognitive practices which can prohibit student progression in such courses.

This paper aims to contribute to this gap by presenting a conceptual software development process which utilities the affordances of computational thinking to create a software development process that encompasses a learning process. The process combines current research into computational thinking as a problem solving process underpinned by the focus of threshold concepts and an Agile philosophy to support students learning how to develop software solutions from problem specification through to the final tested product. The aim of the process is to provide scaffolding to students as they learn how to develop software in a systematic fashion. It is the contention of this research that the provision of such a process could provide a structured and scaffolded environment to directly address the maladaptive cognitive habits that students often form and find hard to unlearn. The next stage of this research will involve the development of a support tool and the deployment and evaluation of the software development process.

References

1. Boehm, B. *A view of 20th and 21st century software engineering.* in *Proceedings of the 28th international conference on Software engineering.* 2006. ACM.
2. Caspersen, M.E. and Kolling, M., *STREAM: A First Programming Process*. Trans. Comput. Educ., 2009. **9**(1): p. 1-29.
3. Kazimoglu, C., Kiernan, M., Bacon, L., and MacKinnon, L., *Developing a game model for computational thinking and learning traditional programming through game-play*, J. Sanchez and K. Zhang, Editors. 2010, AACE: Chesapeake, USA. p. 1378-1386.
4. Liu, C.-C., Cheng, Y.-B., and Huang, C.-W., *The effect of simulation games on the learning of computational problem solving.* Computers & Education, 2011. **57**(3): p. 1907-1918.
5. Xiaoyuan, S., *Toward more effective strategies in teaching programming for novice students.* Teaching, Assessment and Learning for Engineering (TALE), 2012 IEEE International Conference on, 2012: p. T2A-1-T2A-3.
6. Coffey, J.W., *Relationship between design and programming skills in an advanced computer programming class.* J. Comput. Sci. Coll., 2015. **30**(5): p. 39-45.
7. Kokotovich, V., *Problem analysis and thinking tools: an empirical study of non-hierarchical mind mapping.* Design Studies, 2008. **29**(1): p. 49-69.
8. Fornaro, R.J., Heil, M.R., and Tharp, A.L., *What Clients Want - What Students Do: Reflections on Ten Years of Sponsored Senior Design Projects*. 19th Conference on Software Engineering Education & Training (CSEET'06), 2006: p. 226-236.

9. Huang, T.-C., Shu, Y., Chen, C.-C., and Chen, M.-Y., *The development of an innovative programming teaching framework for modifying students' maladaptive learning pattern.* International Journal of Information and Education Technology, 2013. **3**(6): p. 591.
10. Loftus, C., Thomas, L., and Zander, C., *Can graduating students design: revisited*, in *Proceedings of the 42nd ACM technical symposium on Computer science education.* 2011, ACM: Dallas, TX, USA.
11. Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J., *A survey of literature on the teaching of introductory programming.* ACM SIGCSE Bulletin, 2007. **39**(2): p. 19.
12. 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.* 2013. ACM.
13. Gautier, M. and Wrobel-Dautcourt, B., *artEoz-dynamic program visualization.* ISSEP 2016, 2016: p. 70.
14. Mozelius, P., Shabalina, O., Malliarakis, C., Tomos, F., Miller, C., and 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*. 2013. Academic Conferences International Limited.
15. Trevathan, M., Peters, M., Willis, J., and Sansing, L. *Serious Games Classroom Implementation: Teacher Perspectives and Student Learning Outcomes*. in *Society for Information Technology & Teacher Education International Conference*. 2016.
16. Wright, D.R. *Inoculating Novice Software Designers with Expert Design Strategies*. in *American Society for Engineering Education*. 2012. American Society for Engineering Education.
17. Hu, M., Winikoff, M., and Cranefield, S., *A process for novice programming using goals and plans*, in *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*. 2013, Australian Computer Society, Inc.: Adelaide, Australia.
18. Neto, V.L., Coelho, R., Leite, L., Guerrero, D.S., and Mendon, A.P., *POPT: a problem-oriented programming and testing approach for novice students*, in *Proceedings of the 2013 International Conference on Software Engineering*. 2013, IEEE Press: San Francisco, CA, USA.
19. Morgado, C. and 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*. 2012, ACM: Haifa, Israel.
20. Siegfried, R.M., Greco, D., Miceli, N., and Siegfried, J., *Whatever happened to Richard Reid's list of First Programming Languages?* Journal of Information Systems Education, 2012. **10**(4): p. 7.
21. Maslow, A.H., Frager, R., Fadiman, J., McReynolds, C., and Cox, R., *Motivation and personality*. Vol. 2. 1970: Harper & Row New York.
22. Higgins, C., Mtenzi, F., O'Leary, C., Hanratty, O., and McAvinia, C., *A Conceptual Framework for a Software Development Process based on Computational Thinking (In Print).* in *11th International Technology, Education and Development Conference.* 2017: Valencia, Spain.
23. Polya, G., *How To Solve It.* 2nd ed. 1957: Princeton University Press.
24. Wirth, N., *Program development by stepwise refinement*. Communications of the ACM, 1971. **14**(4): p. 221-227.
25. Pizka, M. and Bauer, A. *A brief top-down and bottom-up philosophy on software evolution.* in *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of*. 2004. IEEE.
26. Liikkanen, L.A. and Perttula, M., *Exploring problem decomposition in conceptual design among novice designers*. Design studies, 2009. **30**(1): p. 38-59.

27. Robins, A., Rountree, J., and Rountree, N., *Learning and Teaching Programming: A Review and Discussion.* Computer Science Education, 2003. **13**(2): p. 137-172.
28. Abelson, H. and DiSessa, A.A., *Turtle geometry: The computer as a medium for exploring mathematics.* 1986: MIT press.
29. Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y., *Scratch: Programming for All.* Communications of the ACM, 2009. **52**(11): p. 60 - 67.
30. Thevathayan, C. and Hamilton, M. *Supporting diverse novice programming cohorts through flexible and incremental visual constructivist pathways.* in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education.* 2015. ACM.
31. Vygotsky, L., *Interaction between learning and development.* Readings on the development of children, 1978. **23**(3): p. 34-41.
32. Bustard, D., Wilkie, G., and Greer, D. *The maturation of agile software development principles and practice: Observations on successive industrial studies in 2010 and 2012.* in *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the.* 2013. IEEE.
33. Kastl, P., Kiesmüller, U., and Romeike, R. *Starting out with Projects: Experiences with Agile Software Development in High Schools.* in *Proceedings of the 11th Workshop in Primary and Secondary Computing Education.* 2016. ACM.
34. Li, C.L., Yang, L.P., and Wang, W. *Application of mind mapping to improve the teaching effect of Java program design course.* in *Computing, Control, Information and Education Engineering: Proceedings of the 2015 Second International Conference on Computer, Intelligent and Education Technology (CICET 2015), April 11-12, 2015, Guilin, PR China.* 2015. CRC Press.