RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

Experience Report on Soft and Project Skills Building through Repetition

Devroey, Xavier; Amrani, Moussa; Vanderose, Benoit

[Link to publication]

Citation for pulished version (HARVARD):
Devroey, X, Amrani, M & Vanderose, B 2021, Experience Report on Soft and Project Skills Building through Repetition. in A Vescan, C Serban, J Henry & U Praphamontripong (eds), *EASEAI 2021 - Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence, co-located with ESEC/FSE 2021.* EASEAI 2021 - Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence, co-located with ESEC/FSE 2021, ACM Press, Athens, Greece, pp. 9-14. https://doi.org/10.1145/3472673.3473959

Experience Report on Soft and Project Skills Building through Repetition

Xavier Devroey
x.d.m.devroey@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Moussa Amrani
moussa.amrani@unamur.be
Faculty of Computer Science,
Namur Digital Institute (NaDI),
University of Namur
Namur, Belgium

Benoît Vanderose
benoit.vanderose@unamur.be
Faculty of Computer Science,
Namur Digital Institute (NaDI),
University of Namur
Namur, Belgium

ABSTRACT

Acquiring soft and project skills during their studies is of paramount importance for computer science students to integrate large development teams after graduating. Project-oriented learning offers interesting opportunities for teachers to tutor students, and allows them to acquire and train those skills in addition to the core topics of the course. However, since most existing curricula require courses to be as independent as possible (for organizational reasons for instance), some topics are covered in different courses in slightly different ways. This repetition is interesting for understanding difficult notions appropriately, but may also hamper students' understanding when closely related concepts are embedded in different ways. We report here on our teaching approach: we propose a series of projects that share a common theme, in order to (i) provide a transversal understanding of common notions seen in separate courses, and (ii) introduce soft and project skills in a progressive way, enabling students to iteratively experience and learn skills that are necessary for professional life. We report on the results of interviews conducted with the students and extract valuable lessons for reproducing this approach in different curricula.

CCS CONCEPTS

• **Social and professional topics → Software engineering education**.

KEYWORDS

software engineering education, project skills, soft skills

1 INTRODUCTION

Beyond the various notions studied during a full Computer Science curriculum, the practice of Software Engineering and Computer Science requires *soft skills* and *project skills* that are expected from students after graduation to integrate development teams of large projects [1, 6, 27]. Among others, teamwork requires communication skills, domain and technical analysis, as well as mastering code version control and tasks automation (*e.g.,* testing and code quality measuring, version control, continuous integration, *etc.*). Adopting project-oriented learning [20, 24] provides an adequate playground for introducing such skills, giving students an early intuition of the challenges and technologies, before studying those topics more extensively.

Typically, Computer Science curricula [7] are decomposed into a *Bachelor*, that focuses on fundamental topics for programming (algorithms and data structures, imperative and object-oriented programming, databases, *etc.*) and on acquiring general-purpose knowledge (mathematics, economy, *etc.*); and a *Master* that includes advanced software engineering topics. Nowadays, universities also emphasize project-oriented learning to enhance the understanding of the targeted topics, train critical thinking, provide realistic case studies that go beyond toy examples, and prepare student for their professional life in industry. However, these projects tend to consume a significant time from professors and teaching assistants to design and test before submitting them to students, then to supervise students during the semester (answering theoretical questions as well as technical ones), and finally to grade the students' work.

Another difficulty is that courses are often required to be independent of each other, or at least loosely coupled, to allow students to customize their study program. As a consequence, some topics are covered in different courses in slightly different ways. This repetition is interesting as it allows students to understand difficult notions [8] from different angles. However, this repetition might also hamper the students' understanding when closely related notions are embedded in different courses and projects in different ways, without making *explicit* the links between the underlying fundamental concepts. For example, the notion of concrete syntax may be presented graphically in a modeling course, but textually in a language theory course, both ultimately relying on an abstract syntax.

This paper reports on a teaching experience of proposing a series of two projects that share a common theme, followed by a larger development project that relies on the same technological framework, in order to (i) *provide a transversal understanding of*

common notions seen in separate courses from different perspectives, and (ii) *introduce soft and project skills in a progressive way*, enabling students to iteratively experience and learn skills, and ultimately ingrain practices that are required in professional life. More specifically, the two first projects happen during the last year of the Bachelor program in Computer Science of the *University of Namur* in Belgium, as part of a *Analysis and Modeling* course, and a *Programming Languages Theory* course. The larger development project is spread over three months of the first year of the Master program as part of the *Software Engineering* course.

For assessing the adequacy of our approach regarding the practical use and the (perceived) impact, we conducted interviews with Bachelor students after the first project as well as during the second project, and with Master students after the more significant project, from which we extract valuable lessons for reproducing this scenario in different curricula. Our initial insights indicate that although the transversal approach is not always obvious for students, the improvements observed in the three courses and the positive feedback denote a positive impact of the approach and motivate us to pursue in the integration and alignments of the projects.

The rest of the paper is organized as follows: Section 2 details the objectives of the courses our series of projects illustrate, in terms of core Computer Science notions, as well as soft/project skills they require. Section 3 describes the projects proposed in each course, as well as their synergies and progressive structure. Section 4 discusses the insights and lessons learned, and Section 5 concludes with final remarks.

2 CONTEXT

The ACM Computing Curricula (2020) [7] identifies the *competencies* that should be developed by students during a Bachelor and a Master in various Computer Science disciplines (*e.g.,* software engineering, data science, cybersecurity, *etc.*). It formally structures competencies by clarifying which *knowledge* (the *know-what*), which skills (the *know-how*) and which *dispositions* (the *know-why*) are observable for accomplishing a given task [7]. For each discipline, a curriculum defines the list of competencies that it will develop with students.

The first, *knowledge*, includes elements both from computing (such as systems modeling, software fundamentals, software development, *etc.*), but also from foundational and professional perspectives (analytical and critical thinking, communication and presentation, problem solving, *etc.*). Those elements may be further refined in subelements, whose discussion goes beyond this paper's scope. The second, *skills*, describes how knowledge is concretely applied through six cumulative levels described in Table 1 (based on the revised Bloom's taxonomy of educational objectives [2]). The last one, *dispositions*, covers habitual inclinations of an individual to apply knowledge and use skills to perform a task (for instance, being adaptable, collaborative, inventive, proactive, *etc.*)

Various knowledge elements can be addressed with different levels of skills. Table 2 provides an overview of how the *Analysis and Modeling*, the *Programming Languages Theory*, and the *Software Engineering* courses cover the main elements for computing, foundational and professional knowledge with the corresponding skills level. Based on our experience and the curricula in Computer

Science and Software Engineering at the *University of Namur*, we indicate the desired skill level for the corresponding knowledge element, as well as the emphasis put by the projects on that element. The next section details how we implemented those knowledge and skills.

2.1 Analysis and Modeling Course

At the end of the course, students are expected (i) to develop *abstraction capabilities* sufficient for analysing expertise domains; and (ii) to express the result of their analysis, *i.e.,* to *model* their understanding, in an implementation-independent fashion, using the industry standard UML, covering both structural (through Class and Object Diagrams as well as OCL) and behavioural (with Use Case, State Machine, Activity and Interaction Diagrams) concerns [11, 12, 17]. A typical exercise consists, for instance, in analysing the description, in natural language, of a reasonably large information system specification. The course focuses on the ability to distinguish which aspects of the domain are adequately covered by which UML diagram(s), while offering a lightweight methodology to approach the analysis of the requirements, and translate into an UML specification.

After noticing that students considered this course difficult due to the complexity of variety of diagrams, we introduced a *modeling project*, whose purpose is to study more deeply the structural aspects of a domain on a more realistic system. The project's theme describes a Domain-Specific Language (DSL): for the two first years, we proposed a wristwatch product line with adaptable apps (inspired by [14]), before switching to a world game [5, 26] to better catch the students' interest. All DSL case studies explicitly integrated a structural, as well as a dynamic part describing transformations on the domain (typically for the world games, a small strategy language). The project's evaluation relies on three criteria: (i) the precision and accuracy of the diagram, tested using predefined, representative witness instances; (ii) the consistency between diagrams (what is formally known as model conformance [4, 16]), and constraints; and (iii) the overall quality of their solution.

Beyond learning the modeling tool (*via* tutorial videos and internet content), completing the project does not imply designing large diagrams. Instead, it requires confronting the students' understanding of the domain, and designing a good working strategy to ensure that the different diagrams are consistent and allow to model the situations adequately.

2.2 Programming Language Theory Course

The course covers the basics of language and compiler theory (regular expressions, context-free languages, static and dynamic semantics, and code generation). *Compiler projects* [15, 19, 25] help students to practice these concepts. In previous versions of the course, our compiler project was following a *software project strategy* [32] and consisted of a simple imperative program compiled in P-CODE [23] (executed using an abstract graphical P-Code Machine [31]), implemented in C and using Flex and Bison to generate a parser from an attributed grammar.

A test-driven approach allowed to ease (auto-)evaluation and assess the progression of the students [18]: they submitted their code to an automated script that ran the code against a series of test

Table 1: Cumulative levels of skills based on Bloom's taxonomy [7]

Remembering	Understanding	Applying	Analyzing	Evaluating	Creating
Exhibit memory of previously learned materials by recalling facts, terms, basic concepts, and answers.	Demonstrate understanding of facts and ideas by organizing, comparing, translating, interpreting, and giving descriptions.	Solve problems in new situations by applying acquired knowledge, facts, techniques, and rules in a different way.	Examine and break information into parts by identifying motives or causes; make inferences and find evidence to support solutions.	Present and defend opinions by making judgments about information, validity of ideas, or quality of material.	Compile information together in a different way by combining elements in a new pattern or by proposing alternative solutions.

Table 2: Computing, foundational and professional knowledge [7] covered by the courses with the corresponding skill level. A ★ indicates that the knowledge element and skill level are emphasised in the course's project.

Knowledge element	Analysis and Modeling	Programming Languages Theory	Software Engineering
Users and Organizations			
Social Issues and Professional Practice			Understanding
Enterprise Architecture			Understanding
Project Management		Applying (★)	Evaluating (★)
User Experience Design			Applying (★)
Systems Modeling			
Systems Analysis and Design	Creating	Evaluating (★)	Evaluating (★)
Requirements Analysis and Specifications	Evaluating		Evaluating (★)
Data and Information Management	Applying		Evaluating (★)
Systems Architecture and Infrastructure			
Parallel and Distributed Computing			Evaluating (★)
Software Development			
Software Quality, Verification and Validation		Analyzing (★)	
Software Process		Applying (★)	Evaluating
Software Modeling and Analysis	Creating	Applying (★)	Evaluating
Software Design	Evaluating	Applying (★)	Evaluating (★)
Software Fundamentals			
Data Structures, Algorithms and Complexity		Analyzing (★)	
Programming Languages		Evaluating	
Programming Fundamentals		Evaluating	
Computing Systems Fundamentals		Analyzing (★)	
Foundational and Professional			
Analytical and Critical Thinking	Applying	Applying	Applying
Collaboration and Teamwork	Applying (★)	Applying (★)	Applying (★)
Multi-Task Prioritization and Management	Applying (★)	Applying (★)	Evaluating (★)
Oral Communication and Presentation			Applying (★)
Problem Solving and Trouble Shooting		Applying (★)	Applying (★)
Project and Task Organization and Planning	Applying (★)	Applying (★)	Applying (★)
Quality Assurance and Control		Applying (★)	Evaluating (★)
Relationship Management			Applying (★)
Research and Self-Starter/Learner		Applying (★)	Applying (★)
Time Management	Applying (★)	Applying (★)	Applying (★)
Written Communication	Applying (★)	Applying (★)	Applying (★)

sets (unknown from the students) and produced a report notifying the number of passed and failed tests in each set. Each test set was a *milestone* that students had to pass before a given deadline. Students were encouraged to design their own tests and to share them with their fellows.

More recently, we decided to make the project more directly relevant for students and their future professional life [9] as well as the core curriculum [10], by emphasizing the practice of soft and project skills. We kept the test-driven approach and updated the technological infrastructure to Java, using ANTLR 4 [21, 22] as a parser generator, Git for version control, and Maven and Jenkins for building and continuous integration.

We also aligned the theme of the project with the one proposed in the modeling project [33]. In the modeling project, students model the domain's structure and constraints (known as static semantics in the compiler project) using UML. In the compiler project, they generate working code for the DSL, starting from a given EBNF grammar description of the structural aspects. This flattens the starting point for all students and preserve independence of both courses.

2.3 Software Engineering Course

During the first year of the Master, this course confronts students to a sizable software development project and put them in a quasi-realistic development context in order to illustrate the challenges facing professional software engineers [6, 13, 24]. It is designed as a capstone course, building upon the knowledge acquired during the Bachelor's years (programming, modeling, *etc.*) and introducing new challenges (requirements engineering, project management, rapid delivery, *etc.*).

This course settings intentionally broaden soft and project skills by facing students with high levels of uncertainty and freedom of choice, which are inherent to real-life software development. It is instrumental in teaching, among others, the importance of communication within teams and the importance of a rigorous rational process to inform the many choices presenting themselves during the course of the development.

Figure 1: View of the playing grid from one player (extending the 2D Roguelike Unity tutorial [28])

```
declare and retain
[...]
when your turn
  when life < 20 do
    next use soda
  done
  by default declare local
    b as boolean;
  do  set b to f()
      next move east
  done
```

Figure 2: Example of B314 program

3 PROJECTS DESCRIPTION

The *modeling project* and the *compiler project*, both part of the third year of Bachelor at the *University of Namur*, share a common theme: it consists of specifying, during the modeling project, a DsL describing a turn-by-turn world game entitled *Live Long and Die Hard!* (LLADH) through a model, expressed as a UML class diagram with OcL constraints. This modeling specification is later on compiled, during the compiler project, into an executable code that allows effectively playing. LLADH is plotted as follows: two explorers, *Steve Intacks* and *Scipion-Edouard Mantics*, compete on different maps where they must retrieve a *Grail* to progress to the next level (see Figure 1 for instance).

LLADH is conceptually built on two small languages: the first describes the world and its components (the maps, collectable items, the Grail, and the persona representing the explorers); the second describes the behavior of the explorers and their strategy to collect the Grail while avoiding attacks from other players. The explorers have a number of primitive actions at disposal to scan the environment on the nearby cells (*e.g.,* is there enemies or items around) and to decide which actions may be taken (*e.g.,* use an item; fight an opponent). To memorize information, the strategy DsL allows defining global, as well as local variables, that can be assigned, and use iterative blocks defined by loops. Figure 2 shows a possible (simple) strategy using the textual representation of the DsL.

3.1 Modeling Project

The project is made available after one month of class during which the course and practicals covers the necessary theory. Students work in pair using *MagicDraw* (and later on, *Modelio*), and are encouraged to discuss their understanding in an online forum where teachers may give precisions about the project. The project has non-compulsory milestones to help students assess their progress. The project description explicitly includes questions to guide the students' work towards an adequate solution, without loosing too much time with the details of the specification. Witness instances, as well as specific constraints, are provided so that students can test that their diagrams cover what is required.

The grading is realized in a classical way: from an ideal solution designed by the teaching team, we assess deviations, missing points as well as correct solutions. Due to time restrictions, detailed feedback to students is provided only after the final evaluation.

3.2 Compiler Project

The project runs in parallel with the course, and starts with a half-day training where we present the DsL describing the behavior of a player, and introduce the different technologies used during the project. We illustrate the different steps required by a compiler (encoding a grammar, validating candidate programs with a typing system, and producing code) through a demo compiler [30] that handles simple arithmetic expressions with variable assignment.

Students then work in trios to develop their compiler, based on a provided specification. To ease the learning progression, the project is divided into milestones, each being associated to a set of compiler functionalities, and a set of corresponding (hidden) test cases. Students have to reach different milestones at given deadlines by passing all the tests associated to the milestone: to pass a test, the compiler shall reject invalid programs, and produce code for valid programs producing the expected output values when executed.

The assessment is automated using a continuous integration server: it provides feedback to students each time they push their code on a *Git* server. We choose to have each milestone as a separate *Jenkins* job, plus a global job that builds, and computes test coverage and code quality metrics of the students' code. After a push, each student group can check which milestones are validated by looking at which jobs succeeded or failed. Grading is based on the succeeded milestones and the quality of the compiler code and tests, and an individual oral exam to assess the global understanding of the different concepts.

3.3 Software Engineering Project

During the 14 weeks of the course, students are randomly divided in teams of around 5 members in charge of a software development project. Each team starts from a loose *elevator pitch* as a project description, and has to elicit the appropriate (non-)functional requirements through interviews with the customers role-played by the teaching team, starting from an initial undisclosed set of precise requirements. All teams follow a customized Agile development method (sometimes referred to as *Water-Scrum-fall* [29]) and are supported by *Jira* Agile tools. The exact business case targeted by the project varies each year but always centers around a distributed application with multiple types of users interacting concurrently.

Except from *Jira* and *SonarQube* for quality control, little to no constraints are put on the development technologies and tools the students have to rely on to achieve their goals. The teaching team is also available as a resource for students when technological or methodological problems arise.

Students are graded, based on the delivered product and its compliance with the requirements. At the end of the semester, they showcase their solution in front of the other groups and members of the faculty, including the teaching team playing customers, and professors with different areas of expertise. During this session, functional and non-functional requirements are demonstrated, followed by a discussion about their relevance as well as the relevance of the architectural and technological choices taken by the teams.

4 LESSONS LEARNED

In April 2018, after all projects in the academic year ended, we evaluated the recent changes introduced in the modeling and programming language theory projects and their reception and helpfulness for the software engineering project, by using two different evaluations. First, we sent an online questionnaire to former students who did the projects in the 2015–2018 period. Although asking students two years later seems to represent a threat to validity, we considered this gave them time to reflect on what those projects bring them in their professional life. We received 60 answers total: with 11 (18.3%) from 2015-2016, 20 (33.3%) from 2016-2017 and 29 (48.3%) from 2017-2018.

Second, we conducted semi-directed interviews: we screened students over the whole spectrum of grades (low-grades, average-grades, and high-grades) in the Bachelor after the compiler project was over, and in the master after the the software engineering project finished. Six students in each category were chosen to ensure obtaining at least one interview per group. We had between two and four students in each category except in high-grade bachelor students and in average and low-grade master students, resulting in a total of 13 interviews.

Both evaluations questioned the difficulty and the utility of the project *w.r.t.* the associated course, and its practical organization. We reflect here on the perception of the transversality and the practice of soft and project skills.

4.1 On the Transversality of the Projects

55% of the respondents perceived the modeling and compiler projects as challenging, while 80% indicated finding them interesting/very interesting, and 45% thought the projects helped/were essential to understanding the theoretical lectures. For the Modeling project, more than half the respondents suggested that projects on other diagrams would be useful (which is unfortunately impossible for a one-semester course). However, other UML diagrams are put in practice later on in the software engineering project. For the Compiler project, a few students suggested developing a tool like *ANTLR* "*to put theoretical concepts into practice.*" It would be interesting, but hard to integrate with a transversal approach, and potentially less motivating.

Among students that passed both the Modeling and Compiler projects (representing 56 respondents and 7 interviewees), 30% considered that having a common theme did not help them start quicker on the compiler project (since it is presented differently: with a metamodel in the Modeling project; and with an EBNF grammar in the Compiler project), while the shared content helped 46% of them better understand the topic and the work expected from them. However, interviews indicate that students still in Bachelor do not see how the Modeling project would help them for other courses.

4.2 On the Soft and Project Skills Perception

52% of the respondents (from a total of 27 answers) indicated that the Analysis and Modeling course and its project prepared them for the Software Engineering project (since they use various UML diagrams studied in the Analysis and Modeling course), while 30% of the students also indicated that the Compiler project prepared them to the technical aspects of coding for the Software Engineering project, and 48% for the testing aspects. Since the proposed witness diagrams were not presented as *tests*, students often submitted work that was not sufficient to handle the object diagrams or the constraints proposed.

Regarding the usage of new technologies in the compiler project, respondents pointed that (compared to the C compiler) "*a DSL with ANTLR is way more interesting and suited to the current state of practice*" and "*it opens more lines of thought in a professional context.*" Other respondents pointed out that the "*technologies are aligned with the professional market*" and "*used daily in my professional context.*"

4.3 Improvements of the Projects and Future Work

From the answers, we observe that the transversal approach is not obvious for the students and that it could be improved on several aspects. However, the positive feedback in the open questions and the improvements observed in the three courses encourage us to continue the efforts for aligning the projects and their topics. This is also underlined by respondents who wish "*a stronger link [with the Modeling project and] with the Compiler project.*" Emphasizing the transversality could be possible by explaining how to bridge (meta-)models and grammars [33] as an exercise during practicals, and by enforcing the link through the use of a tool that automates the bridging (*e.g.,* Xtext [3]).

For the compiler project, test coverage and code quality became part of the final grade after the first year to encourage the students to add tests. We want to continue that effort by providing them with finer grained feedback through the continuous integration server to suggest potential sources of bugs using static analysis tools. It would improve their project skills with minimal effort by merely exposing them to new development tools without requiring heavy training, and better prepare them for the software engineering course.

Following the same idea, development of other skills, like project management or continuous integration, may also be pushed one step further by using GitHub, instead of an local Git server installation, and benefit from its project management capabilities like milestones definition, scrum boards, *etc.* This approach may be easily turned into practice at a low cost for students.

5 CONCLUSION

At the *University of Namur*, we designed two projects illustrating both the Analysis and Modeling, and the Programming Language Theory courses, with a shared topic: students need to analyse, model, and later generate full code for a Domain-Specific Language representing a world game. Although the transversal dimension of both projects is not immediately perceived, students benefit from working on an already modelled Dsʟ. Introducing soft (communication, team management) and project (version control, testing, continuous integration, *etc.*) skills early better prepares the students to handle a more realistic project in a Software Engineering course realised in larger teams.

Our initial insights are encouraging and motivate us to pursue our efforts for integrating and aligning the topics and teaching requirements of those projects. Future work includes the monitoring of the students and the approach in a more systematic way: using evaluation forms and data collected by the development tools (like the continuous integration server) to initiate a iterative improvement of the projects definitions. Furthermore, since the Dsʟ provides a visual representation, it may be interesting to further enforce transversality by integrating knowledge and skills developed in the Human-Computer Interactions course.

ACKNOWLEDGMENTS

REFERENCES

[1] Faheem Ahmed, Luiz Fernando Capretz, and Piers Campbell. 2012. Evaluating the demand for soft skills in software development. *IT Professional* 14, 1 (2012), 44–49. https://doi.org/10.1109/MITP.2012.7

[2] Lorin W Anderson, Benjamin Samuel Bloom, and Others. 2001. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives.* Longman.

[3] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing.

[4] Jean Bézivin. 2005. On The Unification Power of Models. *Software And Systems Modeling* 5 (2005), 171–188.

[5] Nergiz Ercil Cagiltay. 2007. Teaching software engineering by means of computer-game development: Challenges and opportunities. *British Journal of Educational Technology* 38, 3 (2007), 405–415. https://doi.org/10.1111/j.1467-8535.2007.00705.x

[6] Dale Callahan and Bob Pedigo. 2002. Educating experienced IT professionals by addressing industry's needs. *IEEE Software* 19, 5 (2002), 57–62. https://doi.org/10.1109/MS.2002.1032855

[7] Alison Clear, Allen S. Parrish, John Impagliazzo, and Ming Zhang. 2019. Computing Curricula 2020. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* ACM, New York, NY, USA, 653–654. https://doi.org/10.1145/3287324.3287517

[8] Thomas B. Corcoran, Frederic A. Mosher, and Aaron Rogat. 2009. *Learning Progressions in Science: An Evidence-Based Approach to Reform.* Technical Report. Consortium for Policy Research in Education (CPRE). 82 pages. https://doi.org/10.1007/978-94-6091-824-7

[9] Saumya Debray. 2002. Making compiler design relevant for students who will (most likely) never design a compiler. *ACM SIGCSE Bulletin* 34, 1 (mar 2002), 341. https://doi.org/10.1145/563517.563473

[10] Akim Demaille. 2005. Making compiler construction projects relevant to core curriculums. *ACM SIGCSE Bulletin* 37, 3 (2005), 266. https://doi.org/10.1145/1151954.1067518

[11] Gregor Engels, Jan Hendrik Hausmann, Marc Lohmann, and Stefan Sauer. 2006. Teaching UML Is Teaching Software Engineering Is Teaching Abstraction. In *Satellite Events at the MoDELS 2005 Conference. MODELS 2005.* LNCS, Vol. 3844. Springer, 306–319. https://doi.org/10.1007/11663430_32

[12] Daniela Giordano and Francesco Maiorana. 2014. Teaching "design first" interleaved with object-oriented programming in a software engineering course. In *IEEE Global Engineering Education Conference (EDUCON)* (2014). IEEE, 1085–1088. https://doi.org/10.1109/EDUCON.2014.6826243

[13] Daniel Gonzalez-Morales, Luz Marina Moreno de Antonio, and Jose Luis Roda Garcia. 2011. Teaching "soft" skills in Software Engineering. In *IEEE Global Engineering Education Conference (EDUCON)* (2011). IEEE, 630–637. https://doi.org/10.1109/EDUCON.2011.5773204

[14] Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society.

[15] Stefan Kögel, Michael Stegmaier, Raffaela Groner, Stefan Götz, Sascha Rechenberger, and Matthias Tichy. 2018. Developing an Optimizing Compiler for the Game Boy as a Software Engineering Project. In *Proceedings of the 40th International Conference on Software Engineering Software Engineering Education and Training - ICSE-SEET '18.* ACM, 9–12.

[16] Thomas Kühne. 2006. Matters of (Meta-) Modeling. *Software and Systems Modeling (SoSyM)* 5 (July 2006), 369–385. Issue 4. http://dx.doi.org/10.1007/s10270-006-0017-9

[17] Grischa Liebel, Rogardt Heldal, and Jan Philipp Steghofer. 2016. Impact of the use of industrial modelling tools on modelling education. In *Conference on Software Engineering Education and Training* (2016). IEEE, 18–27. https://doi.org/10.1109/CSEET.2016.18

[18] Isabelle Linden, Hubert Toussaint, Andreas Classen, and Pierre-Yves Schobbens. 2008. Automatic Student Coaching and Monitoring Thanks to AUTOMATON: The Case of Writing a Compiler. In *European Conference on e-Learning* (2008), Roy Williams (Ed.). Academic Conferences Ltd, Cyprus, 109–117.

[19] M. Mernik and V. Zumer. 2003. An educational tool for teaching compiler construction. *IEEE Transactions on Education* 46, 1 (feb 2003), 61–68. https://doi.org/10.1109/TE.2002.808277

[20] Julie E Mills and David F Treagust. 2003. Engineering education - Is problem-based or project-based learning the answer? *Australasian journal of engineering education* 3, 2 (2003), 2–16.

[21] Terence Parr. 2010. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* Pragmatic Bookshelf.

[22] Terence Parr. 2013. *The Definitive ANTLR 4 Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf.

[23] Steven Pemberton and Martin Daniels. 1982. *Pascal Implementation.* Ellis Horwood Ltd.

[24] Dragutin Petkovic, Rainer Todtenhoefer, and Gary Thompson. 2006. Teaching Practical Software Engineering and Global Software Engineering: Case Study and Recommendations, In Frontiers in Education Conference (2006). *Proceedings. Frontiers in Education. 36th Annual Conference*, 19–24. https://doi.org/10.1109/FIE.2006.322377

[25] Derek Rayside. 2014. A compiler project with learning progressions. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014.* ACM Press, 392–399. https://doi.org/10.1145/2591062.2591168

[26] G Sindre, S Line, and O.V. Valvag. 2003. Positive experiences with an open project assignment in an introductory programming course. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 608–613. https://doi.org/10.1109/ICSE.2003.1201244

[27] Deborah H. Stevenson and Jo Ann Starkweather. 2010. PM critical competency index: IT execs prefer soft skills. *International Journal of Project Management* 28, 7 (2010), 663–671. https://doi.org/10.1016/j.ijproman.2009.11.008

[28] Unity Technologies. 2015. *2D Roguelike.* Unity Technologies. Retrieved July 2, 2021 from https://assetstore.unity.com/packages/templates/tutorials/2d-roguelike-29825

[29] Georgios Theocharis, Marco Kuhrmann, Jürgen Münch, and Philipp Diebold. 2015. Is Water-Scrum-Fall Reality? On the Use of Agile and Traditional Development Practices. In *Product-Focused Software Process Improvement. PROFES 2015. (LNCS, Vol. 9459)*, Pekka Abrahamsson, Luis Corral, Markku Oivo, and Barbara Russo (Eds.). Springer International Publishing, 149–166. https://doi.org/10.1007/978-3-319-26844-6_11

[30] Hubert Toussaint, Xavier Devroey, and Yves Bontemps. 2014. DEMO, un langage d'exemple.

[31] Hubert Toussaint, Xavier Devroey, Yves Bontemps, and Andrew Khvalenski. 2008. *GPMachine: a virtual machine interpreting P-Code.* https://doi.org/10.5281/zenodo.5059924

[32] William M Waite. 2006. The compiler course in today's curriculum: three strategies. *ACM SIGCSE Bulletin* 38, 1 (mar 2006), 87. https://doi.org/10.1145/1124706.1121371

[33] Manuel Wimmer and Gerhard Kramler. 2005. Bridging Grammarware and Modelware. In *Workshop in Software Model Engineering (WiSME).* Springer, 159–168.