On the Design of a New Software Engineering Curriculum in Computer Engineering

Stefan Hallerstede(✉), Peter Gorm Larsen, Jalil Boudjadar, Carl Peter Leslie Schultz, and Lukas Esterle

DIGIT, Department of Engineering, Aarhus University, Aarhus, Denmark
{sha,pgl,jalil,cschultz,lukas.esterle}@eng.au.dk

Abstract. The Department of Engineering at Aarhus University has started up a new science-based BSc degree in Computer Engineering. We report about our planning of the curriculum during the first two years in the Software Engineering area. We discuss highlights, basic concepts, selected course contents, inter and intra course progression, observations from the first two semesters taught, and our expectations concerning the learning objectives and outcomes of the curriculum as a whole.

1 Introduction

At Aarhus University (AU) the initial engineering educations originally came from the Engineering College of Aarhus, which was merged with AU in 2012. Thus, the prime focus on the original curriculum was to deliver new BSc students that directly were employable by the many local companies needing new employees with skills in the core technologies used right now. This includes programming environments such as C# and both embedded and Windows-based technologies. The courses of the curriculum we describe here are designed as independent units combining theoretical, methodological and practical aspects, emphasising their orientation towards engineering.

The perspective of local companies is brought into the curriculum design by means of an advisory board where the companies are represented. There, the proposed BSc curricula are reviewed and recommendations given. From a university perspective this curriculum strengthens the profile of the newly established faculty of technical sciences that complements the faculty of natural sciences.

When designing the curriculum we have made assumptions and decisions based on our experience at university teachers and researchers. Structure and content of the different courses is based on tried practices, and first-hand information from students in terms of direct feedback – using questionnaires and end-of-course discussions– and observing learning success – their ability to apply the acquired knowledge and reason about it. The general approach follows [1].

J.-M. Bruel et al. (Eds.): FISEE 2019, LNCS 12271, pp. 178–195, 2020.
https://doi.org/10.1007/978-3-030-57663-9_12

Overview. Section 2 describes the courses of the curriculum and their relationships, as well as, some assumptions and decisions we have made when designing the software engineering courses of the curriculum. In Sect. 3 we present an outline of selected courses to clarify the contents and level of the curriculum. Finally, in Sect. 4 we close with a discussion of some issues surrounding the Software Engineering curriculum and some insights since we have started teaching it.

2 Courses

Among the courses of the computer engineering curriculum we focus on the software engineering courses listed below. There are links to and from courses outside the software engineering scope, that are referred to, e.g., the Software Architecture course refers to the Computer Networks course in the 4th semester and Programming and Modelling refers to Classical Physics in the 1st semester. An overview of these relations is given in Fig. 1 where the software-related courses are shown in green. Although these courses are related to the Software Engineering curriculum, they are not included in the current description because their focus is not software as such. Note, however that courses like Computer Networks also uses C, a preview of which is given in the Software Architecture course. However, too much uniformity across the study would risk forcing alien concepts with a focus on software on courses where this is counterproductive.

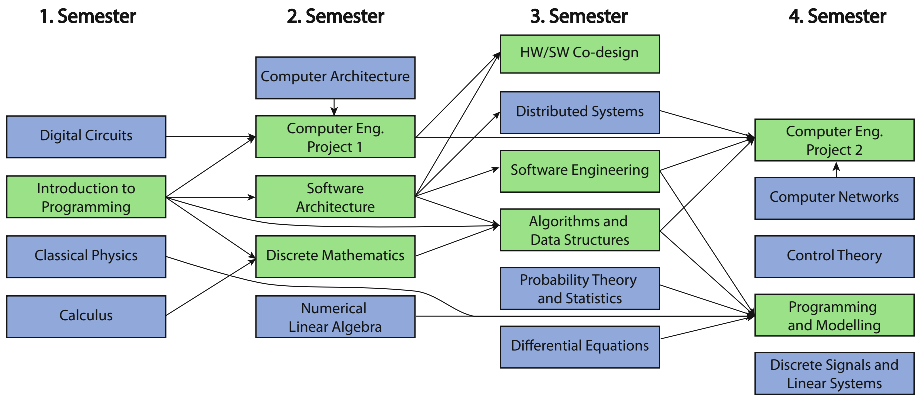


Fig. 1. The first two years of the curriculum for the Bachelor in Computer Software Engineering. Highlighted in green are courses with a focus on Software Engineering. Each column represents a single semester within the first two years of the curriculum. Arrows between the boxes indicate their main dependencies. (Color figure online)

2.1 Variation in Learning Levels

We assume that each cohort of students beginning a BSc study has a wide variety of prior knowledge and talent, and we distinguish them into three major groups (weaker, regular, stronger) for which we cater in the teaching to support the different groups according to their needs. The students are informed about the different offers at the beginning of the different courses. It should be emphasised here that although we consider the students according to the three groups, we make offers to all of them, that range from supporting students with exercises to challenge exercises and projects. The students can make informed choices depending to how they assess themselves at certain points in time. The main outcome of this approach will be a larger spread of talent in Computer Engineering providing as much support as possible to the different learning levels:

Weaker Students: Dedicated to weaker students, support with home assignments is offered in specific exercise sessions.
Regular Students: Regular students receive a large number of exercises to practice their skills, and we will also offer help with solving them, similarly to the weaker students.
Stronger Students: Dedicated to stronger students, we offer challenging voluntary projects that go beyond the course objectives. These projects bring the stronger students in direct contact with research groups.

2.2 Technology and Practical Orientation

The courses will build on software technologies shared between them. The tools used in the courses during the lectures are shared between the courses, although the students are allowed to use their preferred tools. However, they only receive specific support for the tools used in the lectures.

Already in the Introduction to Programming course (see Sect. 3.1) the students are allowed to use modern integrated development environments (IDEs) but we do not use IDEs in the lectures because they hide underlying technologies such as compilation and linking processes. Before using such tools the students should understand the basic functions that are hidden away. IDEs only become necessary when larger programming projects are carried out. Typically, the students do not have problems choosing their preferred IDEs by themselves.

Projects are based on platforms such as Raspberry Pi and Arduino for the deployment of software.

2.3 Languages and Notations

The basic programming language is C/C++ using gcc and gdb that can be installed in Linux, Windows and MacOS. The students receive help installing this software during the first programming café. At the end of the Introduction to Programming course the students receive a short introduction to Python relating it to the concepts they have encountered in C/C++. Python is used in their courses on Numerical Linear Algebra and Classical Physics (see Sect. 2). It is unrealistic to base the entire study on one programming language but we minimise the distractions caused by switching between different programming languages.

This approach permits us to focus on just one programming language to cover the Software Engineering part of the BSc programme, from the presentation of basic algorithms and application development to controller design for embedded systems. Software systems modelling uses graphical notations such as Unified Modelling Language (UML) [5], Architecture Analysis and Design Language (AADL) [3] and formal modelling language the Vienna Development Method (VDM) [4]. All of them have been standardised and a sufficient number of secondary literature is available online and in book form, permitting the students to consult additional sources as it aids their understanding.

2.4 Cross-Cutting Themes and Progression

The courses of the curriculum are connected by underlying themes that span across the two years

- problem solving (PS),
- modelling (MD),
- reasoning (RS),
- verification (VR).

In the first semester we address these connections mostly informally, and as the curriculum progresses towards the fourth semester these connections become increasingly formal. The corresponding skills that students develop are trained and refined from semester to semester. Table 1 shows on which topics and skills the different courses focus. If these were developed in isolation, they would easily end up disconnected from each other, making it difficult for the students to see the big picture of software engineering. This was a major concern when we

Table 1. The software engineering curriculum

Sem.	Lecture	ECTS	Main focus
1	Introduction to Programming	10	Problem solving, programming in C
2	Comp. Engineering Project I	5	Group work, embedded software
2	Software Architecture	5	Software qualities, principles & patterns
2	Discrete Mathematics	5	Abstraction and proof
3	Algorithms & Data Structures	5	Abstraction and efficiency
3	Software Engineering	5	Essential software technologies
3	HW/SW Co-design	5	Programming, modelling, simulation & synthesis
4	Comp. Engineering Project II	5	Planning, organising, distributed software
4	Programming and Modelling	10	Cyber-physical systems modelling, code generation

developed the first drafts of the curriculum: on one hand we have to cover quite distinct topics in a focussed way while the students attending the courses should be able to make connections themselves. We help this by giving ample indications of related courses (and how) and the intended progression during the lectures. As a consequence, the students develop a sense for their overall progression: they respond extremely well on material of this sort in the lectures.

The courses in the curriculum cover different topics of software engineering going deeper in the cross-cutting themes. Table 2 below gives an overview of the relevant themes per course. We introduce concepts for verification like pre- and post-conditions in *Introduction to Programming*, discuss these ideas in *Discrete Mathematics* more formally, consider how it is related to systematic testing in *Software Engineering* and finally combine these techniques in *Programming and Modelling*. As a result, we cover a lot of ground of what could be a course on verification without having it explicitly in the curriculum and, at the same time, provide a greater sense of coherence and continuity for the students.

Table 2. Coverage of cross-cutting themes

Sem.	Course	PS	MD	RS	VR
1	Introduction to Programming	X		X	X
2	Comp. Engineering Project I	X			
2	Software Architecture	X	X		
2	Discrete Mathematics		X	X	X
3	Algorithms & Data Structures	X	X	X	
3	Software Engineering	X			X
3	HW/SW Co-design	X	X		
4	Comp. Engineering Project II	X	X	X	
4	Programming and Modelling	X	X	X	X

The different themes are practiced until they are all used in the course Programming and Modelling in the 4th semester introducing the students to a multi-disciplinary setting around Cyber-Physical Systems [9]. This approach permits to teach advanced material at that stage in the curriculum without having explicit introductory courses on formal methods and related topics. An additional advantage is that the students see the four themes as being related to the topics of the different courses from the start. For instance, the students will learn that programming always involves some form of reasoning about why the program "works", arguing how certain functionality is implemented or why a program terminates, in particular, from the beginning of their studies. They will be aware that there are different approaches to this including informal arguments, formal proofs, testing and debugging. They will become engineers that make pragmatic choices depending on problems at hand, and we teach them a range of techniques on which they will be able to make decisions. In particular, we avoid favouring specific techniques so that the students do not get biased by our teaching.

2.5 International Collaboration

The technologies used in the courses partly are freely available, as well as locally developed and maintained. Partly they are developed at other universities and already used for teaching there. We collaborate with researchers at other universities for incorporating their technologies into our courses. At the moment these are, in particular, the following two technologies to be used in the Programming and Modelling course in the fourth semester (see Table 1 for an overview of the courses):

- The University of Bremen: We will use an automated test case generation library (FSM library at https://github.com/agbs-uni-bremen/fsmlib-cpp) that has been developed at the University of Bremen for teaching purposes. It is a scaled-down version of complex automated verification tool developed by Professor Jan Peleska and colleagues.
- Kansas State University (KSU): We will use verification tools developed at KSU [11].

We intend to extend the number of collaborations in the coming years to internationalise the study environment and ensure that the students develop a sense for the international character of the scientific and engineering communities. The collaborations themselves permit us to have access to advanced tools with a strong background in academic research, as well as, to benefit from teaching experience at those universities.

The alignment of these courses into a coherent set of interconnected topics poses a challenge. In addition to a natural progression of core skill sets such as programming, we adopt cross-cutting themes that are followed up in the different courses. As a consequence the connections between the courses go beyond mere references to related and specialised courses. This is a unique opportunity offered when designing new curricula. Adapting an already running curriculum is difficult by comparison as it means changing courses that work well. It is a continuing effort to preserve this structure.

3 Outline of Some Courses

In order to get an impression of the way the courses are taught, we outline four of them,

(1) Introduction to Programming (Semester 1)
(2) Software Architecture (Semester 2)
(3) Discrete Mathematics (Semester 2)
(4) Computer Engineering Project I (Semester 2)

where we discuss the first one in more detail to illustrate how the courses are designed. For the other three we summarise the main content and highlights.

3.1 Course: Introduction to Programming

Following the distinction of the students into three groups (weaker, regular, stronger, see Sect. 2.1), the *Introduction to Programming* course (8 h/week) uses three types of interaction with the students:

- (frontal/interactive) lectures (L),
- exercise sessions with feedback on assignments (E),
- programming cafés with active support (C).

The students receive information in the beginning of the course where we explain our approach transparently to everyone. The following is a summary of the information given in the first lecture concerning the learning environment and our rationale:

- All students should receive support
- We distinguish the learning levels of students: weak, regular, strong
- We offer a programming café every week where all students will receive help with the course material from the current week. In particular, weaker students benefit from such offers. Students get help with tools, review exercises and past exercises. Exercises for every week will have different difficulty levels with a challenge exercise for the strong students (the challenge exercise is not obligatory and open to all).
- Each week there will be a challenge exercise that students are invited to try themselves: don't do it if you are short of time (for whatever reason); don't do it if you are struggling with the regular exercises; do it otherwise and enjoy doing it!
- In week 4 of the course, a voluntary programming project will be offered to all students. We expect that this will mostly be taken on by regular and strong students. The students decide which offers they take up.

We have developed the course curriculum (Table 3) after a review of similar *Introduction to Programming* courses in many universities around the world. Each topic is introduced by first motivating it through *problem solving* in the context of programming:

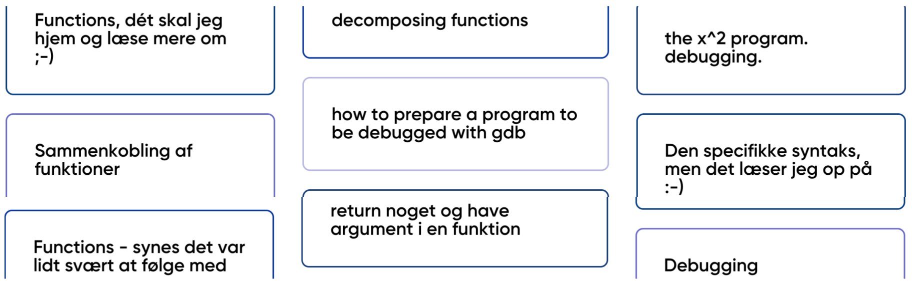
- **aim** e.g. "We want to guarantee that our stack program will work as we intend it to";
- **problem with executable code examples** to concretely illustrate the problem, e.g. stepping through a series of programs that misuse the stack implementation;
- **problem formulation as a more general statement**, referring back to the problem examples, e.g. "The problem is that stack variables are exposed"
- **solution statement**, e.g. "we need to restrict access to variables that define the stack"
- **main topic**, e.g. data encapsulation and object oriented programming
- **problem revisited**, e.g. demonstrating how data encapsulation is used to solve each presented problem example.

Reasoning is first practiced *informally*, arguing for the correctness of small software artefacts, and then *formally* proving correctness of simple programs using a dedicated theorem proving environment for C programs. Topics were repeatedly revisited and referred back to throughout the course in an iterative way, to (a) emphasis the connection *between* topics and (b) give students time and opportunity to really digest and understand each topic through repetition in slightly different contexts.

Table 3. The Introduction to Programming lectures

Week	Topic	Theme	Contact Hours
1	Basics	Programming, problem solving, and reasoning	4L + 4C
2	Loops	Development and derivation of (simple) programs	4L + 4C
3	Testing	Development and analysis of correct programs	4L + 2C + 2E
4	Problem Solving	Array data structures	4L + 2C + 2E
5	Data Structures	Structured data and pointers	4L + 4C
6	Pointers to pointers	Programming with pointers	4L + 2C + 2E
7	Library modules	Programming larger software projects	4L + 2C + 2E
8	Algorithms	Designing sequences of program instructions for solving problems	4L + 2C + 2E
9	Recursion	Functions that call themselves	4L + 4C
10	Higher-order programming	Passing functions as arguments to other functions	4L + 2C + 2E
11	Problem Solving with Recursion	Verifying correctness of recursive programs	4L + 2C + 2E
12	Object-Oriented Introduction	Bundling data and their functions together	4L + 2C + 2E
13	Object-Oriented Programming Constructs	Controlling object state and access to internals	4L + 2C + 2E
14	Python & Recap	Introduction to Python	4L + 2C + 2E

We use an official course textbook [6] to provide students with further background reading and additional practice exercises outside of lectures (although the structure of our course differs significantly from the structure of the textbook). Lecture slides were developed with sufficient detail to be "self-contained" so that they also function as lecture notes, and are provided in PowerPoint and PDF format to students before the lecture; in total we produced over 1000 slides for the course, in 13 lectures (Table 3). In addition, *slidecasts* were created during the lectures and made available after the lecture until the end of the exam



Functions, dét skal jeg hjem og læse mere om ;-)

decomposing functions

the x^2 program. debugging.

Sammenkobling af funktioner

how to prepare a program to be debugged with gdb

Den specifikke syntaks, men det læser jeg op på :-)

return noget og have argument i en funktion

Functions - synes det var lidt svært at følge med

Debugging

Fig. 2. Live student feedback in response to the question "What were the most muddy aspects today?" collected using *mentimeter* and presented back to the students for discussion.

period i.e. recording audio from the lecture, and recording the lecturer's screen that presents the slides.

The initial size of the cohort in 2019 was 30. In Autumn 2020 this will be increased to 60. During the lecture we invite live feedback from the students (2–3 times per lecture) on their understanding of the presented topics by answering 3–4 content related questions via *mentimeter* (Fig. 2).[1] The (anonymous) answers and statistics of the responses are shown on the lecture slide and used as a focused discussion point to clarify the topic at hand.

The lectures are *interactive* in that students are able to execute code presented on the lecture slides. All code examples are made available as C programs (downloadable as a zipped folder) that can be compiled and executed, with the program file name always listed on the lecture slide. To avoid issues with differences in C compilers between operating systems, we created a Virtual Box image with all compiler and debugging tools already installed.

Assignments and Challenge Project. Home assignments are given weekly with sets of problems for all students and a *challenge* problem for the stronger students. The assignment sheet is made available after the lecture on a Tuesday (Fig. 3), and students are required to submit their solutions on the following Tuesday. On Thursday we run a four hour *programming café*, a friendly lab environment in which students can work on their assignment and exercises from the textbook with two teaching assistants available for discussion, input, etc. This approach permits us to give more tailored support for the three groups

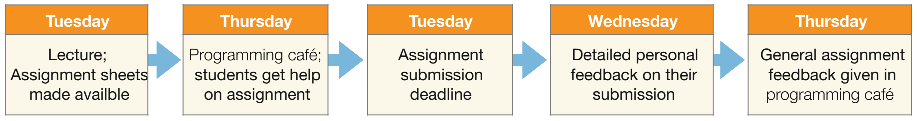

Tuesday
Lecture; Assignment sheets made availble
Thursday
Programming café; students get help on assignment
Tuesday
Assignment submission deadline
Wednesday
Detailed personal feedback on their submission
Thursday
General assignment feedback given in programming café

Fig. 3. Schedule for one assignment (*Introduction to Programming*).

[1] https://www.mentimeter.com/.

(weaker, regular, stronger) of students. One day after submission, students get a grade of either *pass* or *"more work needed"* (allowing resubmission in the subsequent weeks), with detailed personal feedback. In the following programming café one of the teaching assistants presents general feedback and common issues to the class for about 10–15 min.

In addition, a semester-long *Challenge Project* is offered to the students (intended for the stronger students but open for all). This is a programming project to be tackled by students in small self-formed groups of around 3–5 students. Each week an instructor hosts a meeting with all the challenge project students together, and discusses the project concepts, questions and progress, in a fun, friendly and informal setting. On the first iteration of the course we had 8 students (out of 28 students in the class) in two groups. This enabled us, and enthusiastic students, to get to know each other, interact and engage early on in their BSc program. The challenge project was specifically to create an interpreter for a subset of the FORTH programming language.

Assessment. The assessment at the end of the semester is by a 3 h handwritten exam without any support graded according to the Danish 7-point scale. During the semester students are required to submit and receive a "pass" grade for their weekly assignments (described above) in order to be able to go to the exam.

Learning Outcomes of the Course. The learning outcomes for this course describe *hands-on* skills related programming concepts and reasoning, the two main factors to support the course's view of "*programming as problem solving*". At the end of the course, the participants are be able to:

- describe and discuss commands and control structures of imperative programming;
- understand the relationship between iteration and recursion;
- describe and discuss structuring mechanisms in different programming styles;
- implement their own programs using different programming styles;
- explain the concept of imperative and functional programming;
- describe assertional techniques for reasoning about programs; and
- reason informally about programs and relate this to tests.

3.2 Course: Software Architecture

The second semester course on Software Architecture is brought to the students as a continuation of the *Introduction to Programming* course. Whereas the latter course focuses on functional requirements for software, the Software Architecture course focuses on non-functional requirements. The proximity of the two courses permits to make this aspect very explicit, showing them two major concerns of programming: writing program code and organising it.

The course is taught in a standard format of two hours of lectures and two hours of exercises every week during which the students present and discuss their solutions to home assignments. A hands-on textbook is used as main text [10].

Table 4. The Software Architecture lectures

Week	Topic	Practice	Patterns	Principles	Background	Programming
1	Introduction					X
2	Language Abstraction				X	X
3	SOLID Principles			X		X
4	Design Patterns		X			X
5	Architectural Styles		X			X
6	Networking Abstraction				X	X
7	Concurrency Design Patterns		X			X
8	Network Architectural Styles		X			X
9	Software Design			X		X
10	Software Metrics			X		X
11	Software Specification			X		X
12	Software Reuse			X		
13	Application Development	X				
14	Summary and Recap					

In weeks 12 and 13 of the Introduction to Programming course the students learn basics about object-oriented programming. This is continued in the Software Architecture course emphasising programming methodology (using C++). The students learn about object-oriented concepts such as inheritance, polymorphism and genericity. However, just like in the Introduction to Programming course this is always embedded into problem solving. Each lecture relates abstract architectural concepts to concrete programming concepts that can be used to realise the abstract concepts. The software is modelled abstractly using graphical notations like UML, T-diagrams (for composing compilers and interpreters) and ad-hoc diagrams. Subsequently, appropriate implementation techniques are discussed. This makes it possible for the students to apply software architecture immediately based on their first-semester knowledge. The students grow their repertoire of problem-solving techniques to scale to larger problems (which they are told about in the first lecture).

Table 4 shows an overview of the lectures of the course.

Because the course takes place in the second semester, there are two lectures where computer engineering *background* is given: the second week discusses compilation, interpretation and languages, and the third week network technology emphasising the abstractions they provide for software development. The networking lecture is held as a guest lecture by the same lecturer who teaches the computer networks course in the fourth semester. The intention of the guest lecture is to provide a sense of continuity to the students that spans the curriculum. A collection of lectures discusses *principle* underlying architectural design, in particular, SOLID [10] and common design principles like “reduce coupling” and “program defensively”. The discussion of specification and reuse go into depth with respect to the Liskov substitution principle and the idea of refinement. About a third of the course discusses *patterns* describing typical elements of software architectures. About three quarters of the course is dedicated to evolving the programming skills from the level of the Introduction to Programming course to large scale software applying the techniques taught in the course. In a final lecture called Application Development the different principles and patterns are applied to embedded, mobile and desktop software. This is complemented by a guest lecture from a local software development company.

3.3 Course: Discrete Mathematics

The second semester course on Discrete Mathematics takes the informal reasoning from the Introduction to Programming course and adds formality to it and discusses alternative strategies for arguments [2]. Discrete mathematics provides the theoretical foundation for programming. It provides mathematical models for common abstractions referred to in programming. It provides the basis on which the (theoretical) performance of programs can be judged. It permits us to make statements about properties of programs. The course will introduce first-order logic, numbers, sets, sequences, relations and graphs, their applications and techniques of proof. An overview of the course can be seen in Table 5.

Table 5. The Discrete Mathematics lectures

Week	Topic	Concepts	Proof	Programming
1–2	Introduction direct proofs and contradiction		X	
3–4:	Counterexamples and proof by contraposition		X	
5	Logic	X	X	
6	Set theory	X	X	
7	Relations	X	X	
8	Functions	X	X	
9–10	Recursion and Induction	X	X	X
11–12	Sequences and recurrence relations	X	X	
13	Graph theory	X	X	
14	Evaluation and exam preparation			

This course is designed with special principles in two separate dimensions. Firstly this course is delivered as flipped classroom where there is limited time spent on presentations of the material in the four hours of confrontation time every week (instead the students need to work with the material themselves outside class with both videos and the text book, while the remaining parts of confrontation time is spend on the lecturers assisting the students with workshops). Secondly this course is organised such that there is an oral exam where

75% of the grade is based on the students ability to present a subject in the curriculum and 25% on the students ability to critically review another student's oral presentation. Both of these principles are introduced here in order to strengthen the students abilities to work with reasoning in an independent manner and to judge where the right level of formality is.

3.4 Course: Computer Engineering Project I

The second semester course Computer Engineering Project I offers a hands-on experience with solving a comprehensive problem where students to reason, combine and apply the knowledge they learn throughout first and second semester courses. This course also offers the opportunity for students to develop new skills related to design space exploration and code optimisation. On the application side, students learn to manipulate individual technologies such as range finder sensors and light sensors to monitor an environment, Arduino boards to process collected data, Raspberry Pi platforms to actuate mechanical components. This permits the students to see the larger context in which software development typically takes place. An integration of these technologies is performed on a Turtle Bot3 Robot simulating a rescue lab where students are introduced to a set of ROS functions. The main goal is to deliver an optimal exploration plan minimising the robot effort to explore an arena and maximise the number of found "victims" to be rescued. Using their knowledge about compiling and execution, students optimise their code to improve the robot response time and reduce the memory use.

Table 6. The Computer engineering project workshops

Week	Topic	Technology	Programming
1	Lab Introduction, system architecture, subversion		X
2	Assembling Arduino and Breadboard	X	
3	Proximity sensors to Arduino	X	
6	Light sensors to Arduino	X	
7	Actuations		X
8	ROS seminar		X
9	Connecting Arduino to Raspberry Pi	X	
10–11	Robots navigation implementation		X
12–13	Optimization		X
14	Competition, demo and examination		

Regarding the design space exploration, students learn the basis of how to choose design alternatives and how to assess the different designs with respect to a set of possibly conflicting metrics. Towards the end of the semester, the students have to deliver a report documenting their implementations and justifying the design decisions taken throughout the course experience. An overview of the course can be seen in Table 6. The fourth semester project course (Computer Engineering Project II) in comparison follows the one discusses here but is more challenging with respect to group coordination and technological mastery. In particular, it will use material taught in accompanying courses on Control Theory and Networking.

3.5 Summary of the Remaining Courses

Table 7 provides an overview of the remaining courses of the curriculum. We provide only the names of the lectures as the content is mostly well-known.

The Software Engineering course has a special function in the curriculum as it collects and links material from other courses. For instance, the lecture in week 4, Software Design, discusses topics from Introduction to Programming,

Table 7. Remaining courses of the software engineering curriculum

Week	Software Engineering	Algorithms & Data Structures	HW/SW Co-design	Programming & Modelling
1	Software development processes	Basics and Introduction	Computer Engineering	Introduction
2	Requirements Elicitation and Analysis	Implementation of Sequences, Queues and Stacks	HW/SW Co-Design	Basic Technologies for Modelling, Proof and Simulationc
3	Requirements Modelling	Array Searching	Model-based Design	Logics
4	Software Design	Fixed Arrays, Dynamic Arrays, Slices and Iterators	Model-based SW Design	Programming and Proof
5	Version Control	Union Find	Model-based SW Design	Automated Reasoning about Programs
6	Software Quality	Array Sorting	SW Mini Project	Automated Reasoning about Programs
7	Formal Specification	Priority Queue	Model-based HW Design	Modelling Methodology
8	Unit Testing	Sequence and Stream Sorting and Searching	Model-based HW Design	Introduction to INTO-CPS
9	Integration Testing	Search Trees	SW-HW System Synthesis	20-sim Tutorial
10	Performance Requirements	Sets and Dictionaries	Design Space Exploration	Co-simulation & Design Space Exploration
11	Requirements Validation	Matrices	Optimisation and Validation	C Code Generation
12	Formal Verification	Graphs	Final Project	Model Validation & Fault Tolerance
13	Specification, Formal Verification and Testing	Petri Nets	Final Project	Other Approaches: JML
14	Recap	Bitsets / matrices	Summary	Recap & Summary

Software Architecture, and Algorithms & Data structures. This is done expressly in order to give more coherence to the curriculum. The topic of requirements that regularly occurs in other courses is treated systematically. This permits to argue the significance of the topic as such warranting more attention.

The Algorithms & Data structures course follows the problem solving perspective the students are already familiar with from semester one. It focuses on practical aspects using abstractions learned in Discrete Mathematics to reason about problems and algorithms that solve them. Theoretical complexity considerations are discussed and related to practical evaluation of implementation variants.

The course on HW/SW co-design permits the students to understand the specificities of hardware design and of software design, and their similarities, in particular, when done in a suitable framework, such as, System-C.

The fourth semester Programming and Modelling course requires familiarity with the four cross-cutting themes as taught throughout the curriculum. This course is directly linked to local research activities at the Department of Engineering. Whereas reasoning about programs is done informally before, it is treated formally with tool support at this stage. This is complemented by modelling of cyber-physical systems as supported by INTO-CPS [8] and continuous modelling in 20-sim [7]. The course relies particularly on the formal training the students have received in Discrete Mathematics, as well as, Differential Equations and Classical Physics that are taught along side the Software Engineering curriculum. In this respect the students will gain the important insight that software can often not be developed without considering the real world with which it interacts.

4 Discussion

In comparison to a typical curriculum in Computer Science, ours is eminently practically oriented. Although students in Computer Engineering study also more theoretical topics, this has a different focus: Computer Engineering students study necessary theoretical issues in as far as it helps for solving engineering problems, whereas Computer Science students are exposed to the theories as such. For instance, our students encounter functional programming and recursion in Introduction to Programming course indicating suitable reasoning techniques, this is pick up in Discrete Mathematics course where suitable formalised proof techniques are taught. In Computer Science the students attend a course on functional programming where they study the underlying lambda calculus and type theory. Section 4.1 discusses briefly our approach to the relationship of science and engineering in the curriculum.

In order to warrant high quality of the taught courses, we carry out regular evaluations through continual feedback from the students during the courses and collecting data about the courses. We discuss this briefly in Sect. 4.2 and give some first results from the Introduction to Programming course.

4.1 Science and Engineering

The learning objectives of the teaching are, of course, related to the contents of the different lectures. Beyond this we also introduce the students from the beginning to our research activities in engineering science. This makes it possible to offer later in their studies BSc thesis projects closer to ongoing research and strengthen the scientific orientation of their education. For the most part scientific education is treated as the background in the courses, gaining larger weight later in the curriculum. At the end of the BSc studies they have seen some scientific methodology and have applied it in their reasoning. However, through the BSc curriculum (including their BSc theses) they will be guided in that reasoning and the choice of methods. A more independent application of scientific methods is only required in their subsequent MSc studies.

4.2 Evaluation

Starting a new BSc programme is a good opportunity to evaluate effectiveness as we do not have constraints by an established course catalogue. We believe that a culture of systematic evaluation will help to create a strong programme and make a contribution to education research. In order to achieve this, we plan to collect systematically data for all the lectures. E.g., for the Introduction to Programming lecture: Numbers of students present at lectures, exercise sessions and programming cafés, number of students attempting challenge exercises, number of students attempting the programming project, number of students succeeding in the afore mentioned. In addition, we have weekly meetings where we discuss feedback from the students concerning their motivation, learning success, and workload. The aim of this is to determine whether the students get the best possible support according to their abilities. This needs to be fine-tuned permanently.

At the start of the semester, 28 students participated in the course. Of these, 23 students finished the course. The drop-outs happened early in the course. We sent out e-mail to follow up the situation from 3 week on but did not receive a response from the five students. The attendance at the lecture was at least 90% (of 23) at the lecture and a the programming café. However, we found that the café was also used by the students to work on problems from other courses. We did not stop this from happening as it turned out that the students would ask for our help with the Introduction to Programming course when they needed it. This is what had been "promised" to the students in the first lecture. The result of the written exam at the end (A: 3, B: 9, C: 9, D: 1, E: 1) confirms that the support worked well towards achieving the learning objectives.

With respect to the home assignments to be handed in by small groups we made the following observations. None of the groups attempted all challenge exercises. Seven groups attempted at least one challenge exercise, one group attempted five challenge exercises. Eight groups handed in one late assignment, one group two late assignments, and three groups three late assignments. Two groups had two resubmit 2 assignments, and 4 groups one assignment.

Eight students started on the challenge project, one group of three and one group of five. After some initial success (not solving the complete problem) the latter group disbanded. The other group continued. One of the students continues the project in the second semester following the Software Architecture lecture. It appears to be a good idea to propose the challenge project to run over two semesters in the first place because most students stopped because of short term work loads in other courses.

4.3 Concluding Remarks and Evolution

We have outlined the Software Engineering curriculum at the Department of Engineering at Aarhus University, discussed the rationale and provided some examples of concrete courses. Given that we have started teaching in the curriculum since autumn 2019 it is too early to draw any hard conclusions. We have however already learned that the students appreciate the learning environment and the material they are being taught. We believe, that asking them regularly for feedback during the lecture has two major benefits: firstly, we can make improvements while the course is running; secondly, it appears to boost the motivation of the students when they get to play an active role in the shaping of their learning environment by receiving and acting on their feedback. Of course, there are some issues that can only be solved from one instance of the course to the next, concerning, for instance, the order in which some of the material is taught where the feedback that we receive refers to the teaching that is already past.

Acknowledgements. We are grateful for the support and contributions to the preparation of teaching materials carrying our café and exercise sessions by Casper Thule, Tomas Kulik, Christian Møldrup Legaard, Benjamin Salling Hvass, Hugo Daniel Macedo, and Peter Würtz Vinther Tran-Jørgensen.

References

1. Biggs, J., Kum Tang, C.S.: Teaching for Quality Learning at University, 4th edn. McGraw Hill (2011)
2. Cusack, C.A., Santos, D.A.: An Active Introduction to Discrete Mathematics and Algorithms, Version 2.6.4 (2019)
3. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language. SEI Series in Software Engineering. Addison-Wesley (2012)
4. Fitzgerald, J., Larsen, P.G.: Modelling Systems - Practical Tools and Techniques in Software Development, 2nd edn. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK (2009). ISBN 0-521-62348-0
5. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edn. Addison Wesley (2003)
6. Hanly, J.R., Koffman, E.B.: Problem Solving and Program Design in C. Pearson (2016)

7. Kleijn, C.: Modelling and simulation of fluid power systems with 20-sim. Int. J. Fluid Power **7**(3) (2006)
8. Larsen, P.G., et al.: Integrated tool chain for model-based design of cyber-physical systems: the INTO-CPS project. In: CPS Data Workshop, Vienna, Austria, April 2016
9. Larsen, P.G., et al.: Frontiers in software engineering education. In: Collaborative Modelling and Co-simulation in Engineering and Computing Curricula (2020)
10. Martin, R.C.: Clean Architecture A Craftsman's Guide To Software Structure And Design. Prentice Hall (2018)
11. Yi, X., Li, R., Sun, M.: Generating Chinese classical poems with RNN encoder-decoder. In: Sun, M., Wang, X., Chang, B., Xiong, D. (eds.) CCL/NLP-NABD -2017. LNCS (LNAI), vol. 10565, pp. 211–223. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69005-6_18