Assessing Problem-Based Learning in a Software Engineering Curriculum Using Bloom's Taxonomy and the IEEE Software Engineering Body of Knowledge

PETER DOLOG, LONE LETH THOMSEN, and BENT THOMSEN, Aalborg University

Problem-Based Learning (PBL) has often been seen as an all-or-nothing approach, difficult to apply in traditional curricula based on traditional lectured courses with exercise and lab sessions. Aalborg University has since its creation in 1974 practiced PBL in all subjects, including computer science and software engineering, following a model that has become known as the Aalborg Model. Following a strategic decision in 2009, the Aalborg Model has been reshaped. We first report on the software engineering program as it was in the old Aalborg Model. We analyze the programme wrt competence levels according to Bloom's taxonomy and compare it with the expected skills and competencies for an engineer passing a general software engineering 4-year program with an additional 4 years of experience as defined in the IEEE Software Engineering Body of Knowledge (SWEBOK) [Abran et al. 2004]. We also compare with the Graduate Software Engineering 2009 Curriculum Guidelines for Graduate Degree Programmes in Software Engineering (GSwE2009) [Pyster 2009]. We then describe the new curriculum and draw some preliminary conclusions based on analyzing the curriculum according to Bloom's taxonomy and the results of running the program for 2 years. As the new program is structured to be compliant with the Bologna Process and thus presents all activities in multipla of 5 European Credit Transfer System points, we envision that elements of the program could be used in more traditional curricula. This should be especially easy for programs also complying with the Bologna Process.

CCS Concepts: • **Social and professional topics** → **Computing education**; **Software engineering education**; *Model curricula;*

Additional Key Words and Phrases: Problem-based learning, software engineering education, curriculum development

ACM Reference Format:
Peter Dolog, Lone Leth Thomsen, and Bent Thomsen. 2016. Assessing problem-based learning in a software engineering curriculum using Bloom's taxonomy and the IEEE software engineering body of knowledge. ACM Trans. Comput. Educ. 16, 3, Article 9 (May 2016), 41 pages.
DOI: http://dx.doi.org/10.1145/2845091

1. INTRODUCTION

The body of knowledge in computer science and in software engineering is constantly expanding, as witnessed by the 2008 interim revision of the joint IEEE and ACM Computer Science Curriculum from 2001, for example [ACM 2008]. Therefore, great pressure is put on university curricula to fit the many topics within the time frame of 3-year BSc and 2-year MSc programs. At the same time, industry is requesting candidates with skills and competencies beyond the purely technical ones, such as skills in collaboration, communication, project management, and entrepreneurship.

Authors' address: P. Dolog, L. L. Thomsen, and B. Thomsen, Aalborg University, Department of Computer Science, Selma Lagerloefs Vej 300, DK-9220 Aalborg-East, Denmark; emails: {dolog, lone, bt}@cs.aau.dk.

DOI: http://dx.doi.org/10.1145/2845091

Most universities offer a set of core courses plus a number of elective courses to cover the breadth of topics and to allow students to pursue their interests and develop their own profile within the subject range offered by their university. However, this traditional way of curricula design has a tendency to view each topic in isolation, and it may be difficult for students to choose elective topics in such a way that they get a coherent profile (let alone a coherent view on what computer science or software engineering is all about). The deficiencies of the traditional model have been discussed elsewhere at length [Felder et al. 1998], and it is not our goal to engage in this discussion.

However, many university educators are looking for ways to transform or make (smaller) adjustments to their curricula and try out innovative pedagogical approaches, for example puzzle-based learning [Falkner et al. 2010] and Problem-Based Learning [Kay et al. 2000]. The latter, although not so new anymore, is still by many considered the ultimate learning style. However, according to O'Grady [2012], "The penetration of PBL into the average computing curricula is shallow."

Aalborg University features a pedagogical model focused on problem-based, project-oriented, and group-based learning: the so-called Aalborg Model [Barge 2010; Kolomos et al. 2004] (which we shall refer to as the old Aalborg Model for reasons that will become evident). The Aalborg Model has successfully been exported to a number of universities around the world, for example, Mexico, Australia, and France [Kjærsdam 2012], and the university hosts the Aalborg Centre for Problem Based Learning in Engineering Science and Sustainability under the auspices of UNESCO.

In the old Aalborg Model, the study programs are organized into semester topics. Each topic is explored by students in group projects. The students usually divide themselves into project groups of five to seven students in the first five semesters of the bachelor's program and the first two semesters of the master's program. In the last semester of the bachelor's program and the last two semesters of the master's program, the group size is usually two to four students, with a few students choosing to undertake individual projects. Each group of students is assigned a (sometimes shared) working room full time. Each project group is assigned a member of the scientific staff as a supervisor for the project. The projects are further supported by concepts learned from lectures with exercises organized around the semester topics. By doing so, the central student learning activities are focused on performing problem-solving tasks in the group. This engages the students' higher cognitive skills and therefore they should learn more deeply. Furthermore, this learning style promotes collaboration, communication, and project management skills. Each semester is credited 30 European Credit Transfer System (ECTS) points, corresponding to 900 hours of study. In the old Aalborg Model, most semesters are split into four courses of 3 ECTS points each, giving a total course load of 12 ECTS points, and a project contributing 18 ECTS points. The four courses are mandatory and usually there are no elective courses. However, the courses are divided into two categories, SE and PE courses, usually with two of each per semester. SE courses are general courses usually devoted to core topics, and the remaining courses, the PE courses, are project related. This means that the project work takes up the lion's share of students' work, usually about 60% of the students' time excluding the project-related courses, rising to 80% if the two project-related courses are counted as part of the project work. With this much focus on project work, the Aalborg Model has until recently been seen as an all-or-nothing approach.

However, following a strategic decision made in 2009 (partly due to economic and political pressure), the Aalborg Model has been reshaped. Group-based project work still takes up the majority of work for the students, but it has been reduced to about 50%. At the same time, courses have formally been detached from the project work and will take up the other 50% (accounting for 15 ECTS points), and courses have been reshaped into larger units (usually taking up 5 ECTS points); thus, the students will

have fewer but larger courses, usually three courses of 5 ECTS points. Furthermore, partly due to legislation and partly due to pedagogical arguments, some semesters now feature elective courses and the new program is structured to be compliant with the Bologna Process [Union 2010] and thus presents all activities in multipla of 5 ECTS points. We shall refer to this model as the new Aalborg Model [Myrdal et al. 2011].

When the new Aalborg Model was presented, some speculated that it was the end of the Aalborg Model [Spaten and Imer 2011]. We are not that pessimistic, and in this article we stipulate that the current changes will make the Aalborg Model more exportable as it will be possible to pick and match and incorporate elements of the Aalborg Model in more traditional curricula. Furthermore, a recent survey of the experiences with the new model has been carried out [Kolmos and Holgaard 2012]. We make references to this survey and add our own experiences over the past 2 years.

To give readers unfamiliar with problem-based learning an introduction to this pedagogical approach, we first describe the software engineering study programs as they operated in the old Aalborg Model. We give examples of student projects and a short analysis of two semesters: the fourth semester, SW4, of the Bachelor in Software Engineering education and the first semester, SW7, of the master's program. We analyze the program wrt competence levels according to Bloom's taxonomy and compare it with the expected skills and competencies for an engineer passing a general software engineering 4-year program with an additional 4 years of experience as defined in the IEEE Software Engineering Body of Knowledge (SWEBOK) [Abran et al. 2004]. We also compare with the Graduate Software Engineering 2009 Curriculum Guidelines for Graduate Degree Programmes in Software Engineering (GSwE2009) [Pyster 2009]. We then describe the new structure of the software engineering study program at Aalborg University. The new structure was rolled out gradually and actually started in the autumn of 2010. We compare the new program with the old program and more traditional curricula and competence levels according to Bloom's taxonomy. Finally, we discuss our view that the new Aalborg Model will be easier to adopt and adapt in more traditional pedagogical settings.

2. PROJECT-ORIENTED AND PROBLEM-BASED LEARNING BACKGROUND

The pedagogical principles underpinning Problem-Based Learning (PBL) emerged at the end of the 1960s at McMasters University in Canada. The principles were first applied to a curriculum for medical sciences aimed at training general practitioners with a strong focus on a holistic view of humanity and a critical vision on the pedagogical practices of traditional medical training at universities, where medical science had grown into a collection of highly specialized and often unrelated fields.

PBL emphasizes development of analytical, methodical, and transferable skills. The main idea behind PBL is the use of different methods and strategies to reach the needed knowledge by identifying and solving (real) problems. The learning process is organized in such a way that the students are actively engaged in finding answers themselves. The success of the outcome clearly depends on the experiences of the problem solver(s) and thus requires training in identifying and solving problems suitable for facilitating reaching the learning goals.

PBL is based on a constructivist perception of learning and teaching. Learning is the student's individual process of constructing knowledge and meaning, and teaching is the "setting up of a situation from which a motivated learner cannot escape without having learned" [Cowan 1998]. In other words, PBL is student-centered learning, not teacher-centered teaching, hence introducing a new teacher role and a new student-teacher relationship. A guiding principle for PBL is that the students have the responsibility for their own learning.

Somewhat independently of PBL, the notion of Project-Oriented (PO) pedagogical principles started to emerge in engineering education as a response to industry demand for a new competence profile of engineers. In addition to traditional teaching of specialized fields, the industry needed engineers who could work together in teams, often with people from other disciplines; thus, PO learning has a strong focus on teamwork and an interdisciplinary understanding of subjects.

Today it can be somewhat difficult to separate the two pedagogical concepts; what one university practices as PO, another may practice as PBL, and the two pedagogical principles supplement each other, emphasizing different aspects of learning. Project-Oriented Problem-Based Learning (POPBL) is today practiced at a number of universities around the world [Kolmos and de Graaff 2014].

From a student perspective, POPBL means working with real-life problems, which meets the interests of students and therefore enhances their motivation. Additionally, POPBL further develops the students' ability for critical thinking; develops their problem-solving skills and project management skills; improves communication, negotiation, and conflict resolution skills; and strengthens analytical and methodological skills, that is, transferable skills.

Seen from a teacher perspective, POPBL encourages a closer relation between teaching and research—and often teachers learn together with their students.

Finally, from an institutional perspective, POPBL means more motivated students, leading to lower dropout rates. It also means more competent graduates and improved interdisciplinary collaboration between staff members, improved collaboration between the university and industry, and a better match between industry needs and graduate skills [Dahms 2014].

In Denmark, POPBL principles were chosen as the underpinning principles at two new universities created in Roskilde in 1972 and in Aalborg in 1974.

Students seem to appreciate their studies at Aalborg University. In a study among 90,000 students at 550 institutions in 20 countries, conducted in 2008 by the Trendence Institute, Berlin, 90% to 100% of the IT and engineering students assessed Aalborg University to be "Good" or "Very good," which was the highest satisfaction rate of students compared to other Danish and European universities [Trendence Institute 2011]. In 2005, the dropout rate at Aalborg University after 5 years of study was 23%, compared to the national average of 56%, and the average study length for a 5-year master's degree was 5 years, compared to the national average of 6.7 years. Although there has been a major increase in student intake since 2005, doubling the number of software engineering students over a 5-year period, the average study length for a 5-year master's degree at Aalborg university was still 5 years in 2012. Furthermore, the industry appreciates graduates from Aalborg University. In a study conducted by the Danish Engineering magazine *Ingeniøren* in 2004, more than 90% of engineering companies rated the overall quality of the education as "Good" and "Very Good."

3. THE SOFTWARE ENGINEERING STUDY PROGRAM AT AALBORG UNIVERSITY: THE OLD STRUCTURE

The software engineering study program at Aalborg University provides an opportunity to enroll in either a bachelor's or a master's program. The prerequisite for entering a master's program is a completed bachelor's education in a related field.

3.1. Software Engineering Bachelor's Program

The software engineering bachelor's program has a nominal duration of six semesters (180 ECTS points). During the bachelor's program, students are introduced to a wide range of subjects, including Object-Oriented Programming; Object-Oriented Analysis and Design; Software Architectures; Design, Implementation, and Evaluation of User

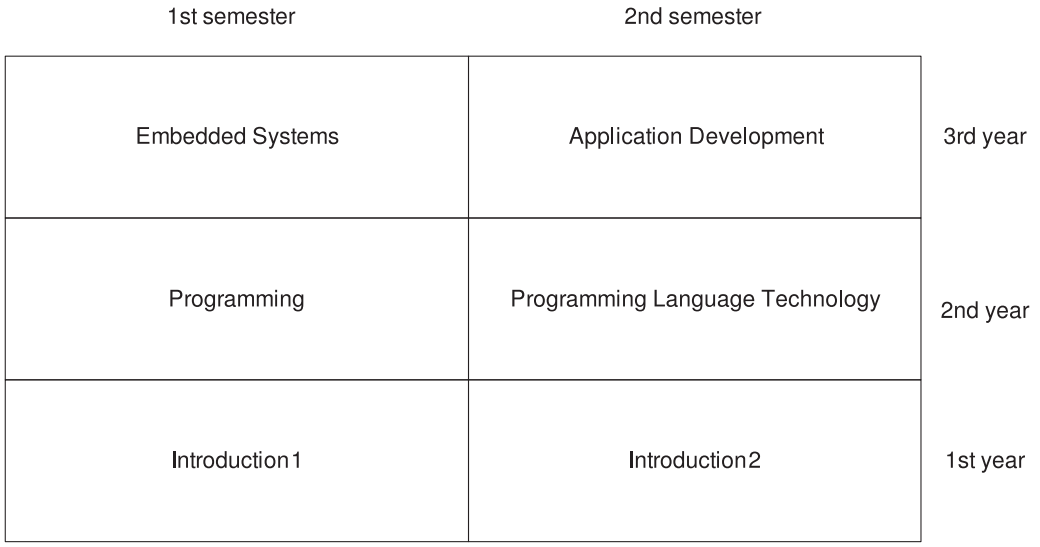

Fig. 1. Software engineering bachelor's program at Aalborg University: the old structure.

Interfaces; and Test and Verification of Software. Figure 1 depicts a schematic structure of the bachelor's study program. Further details are summarized in Appendix B. The first two semesters are devoted to introductory topics necessary for the software engineering education.

The first semester offers students five modules: three course modules and two project modules. Course modules are Linear Algebra; Problem-Based Learning in Science, Technology, and Society; and Imperative Programming. Project modules are "If Programs Are the Solution, Then What Is the Problem?" and "From Existing Software to Models."

The second semester offers students four modules: one project module and three course modules. Course modules are Discrete Mathematics, Computer Architecture, and Object-Oriented Programming. The project module is devoted to Programming and Problem Solving.

The third semester offers students four course modules, one mini project, and one semester project module. The subject of the semester is programming, and therefore, the semester project module is devoted to programming. Course modules are Object-Oriented Programming, System Analysis and Design, Algorithmic and Data Structures, and Software Architecture and Databases. The mini project is devoted to programming.

The fourth semester offers students four course modules and one project module. The semester subject is language technology. The courses offered are Languages and Compilers, Syntax and Semantics, Computer and Network Architecture, and Principles of Real-Time and Control Systems. The project module is devoted to programming language design and translation.

The fifth semester subject is embedded systems. The semester offers students four course modules and one project module. The project module is devoted to design and implementation of an embedded system and associated software. The course modules are Real-Time Software, which looks at a specific software framework for embedded systems; Computability and Complexity; Design and Implementation of User Interfaces; and Programming Paradigms.

The sixth semester subject is application development. The semester offers students five course modules and one semester project module. The project module is the bachelor's project and it is devoted to multiproject application development, where all student groups participate in a large integrated application development. This project is the bachelor's project. The course modules are Test and Verification, Management of Multi Projects, Software Engineering, Database Management Systems, and Professional Communication in Computer Science and Theory of Science.

3.1.1. Programming Language Technology: The Fourth Semester (SW4). To make the study program more concrete, we will in the following look at the contents of the fourth semester.

The subject of the fourth semester is Programming Language Technology with a focus on programming language design, description, and implementation. The main motivation is that programming language technology is at the core of software engineering, and having in-depth insight into how languages are designed, described, and implemented is paramount to a software engineer. This knowledge is essential for understanding existing languages and compilers as well as giving the students the tools and techniques to develop their own languages or extensions of existing languages, which sometimes is the most viable solution.

The main outcomes to be achieved at the competence level are that the students are able to

—demonstrate knowledge and understanding of tools and techniques for defining and implementing programming languages, and
—demonstrate skills in development of runtime and semantic representation of programming language concepts.

Most projects will design, define, and implement a programming language. Usually the language will be a small imperative language designed for a specific purpose, and in most cases a compiler for the language that generates some form of intermediate code, such as Java Byte Code, will be built. Some projects look at declarative languages, and other projects look at generating machine code for a specific architecture; some projects even define and implement their own virtual machine and use that as a target for their compiler.

In general, the students build up their skills in programming as compilers are usually large software artifacts. They also build up skills in analyzing situations that naturally can benefit from the development of programming technology.

As examples of projects we include the students' own descriptions of their projects:

Beginners INtermediate Declarative Graphical Language (BINGLE). "BINGLE is a graphical programming language designed specifically for this report. The first part of the report contains information concerning graphics in general, linguistic theory and compiler creation, which will serve as basis for the rest of the report. The second part focuses on the design process behind BINGLE, including the syntax and semantics of the language. The last part contains a detailed explanation of how BINGLE is implemented, using code extracts as example" [Friis et al. 2006].

Imp: An Implicitly Parallel Language. "Today, systems with multiple processing units are becoming more common. To fully utilize the potential of these systems, programmers have to take parallelism into consideration. This report focuses on how to move the responsibility of parallelism from the programmer to the language processor. To realize this, we analyze the various aspects of compilers and parallel computing. In order to achieve implicit parallelism, we have defined a new programming language called Imp. We have defined semantics and a Context Free Grammar in Extended

Backus-Naur Form for Imp. The implementation of Imp consists of a tokenizer, parser, checker, dependence analyzer and code generator, which are described in detail in this report. Furthermore we have reflected on potential features of Imp if we had additional time to continue the development. We have concluded that it is possible to move the responsibility of parallelism because our product is able to implicitly parallelize a Foreach-loop from Imp to Java" [Bentzen et al. 2009].

A Scripting Language for Mobile Applications. "The theme of this report is language technology. The report documents the development of a scripting language for mobile applications, called Mobile Card Gaming Language (MCGL). The purpose of MCGL is to make it easier for novice programmers to develop card games for mobile telephones. The report describes the processes concerning the design and developing of a language and compiler. SableCC has been used in the development of MCGL, and is described and documented in the report. The first part of the report contains an analysis, providing the basic knowledge of what is required to develop a programming language. The second part provides a description of how the language MCGL will be designed and implemented. Finally, an evaluation part will be used to reflect on the language design and implementation. A tutorial is included, allowing the reader to get a first hand sight of how games can be written in the MCGL language" [Sørensen et al. 2008].

Analysis of Projects. We have analysed 33 projects done in the period between 2004 and 2011. All projects designed a new language; of these, 21 were imperative languages, four object oriented, and eight declarative. Fourteen projects built a compiler by hand without the use of compiler-compiler tools. Four projects used Lex/Yacc (or their Java or C# counterparts), 11 projects used SableCC, and 10 projects used other tools (ANTLR, GOLD parser, or JavaCC). Ten projects generated code for the JVM or the CLR. Three projects created their own Virtual Machine and used that as a target. Nine projects generated machine code, and 11 projects generated code in a high-level language, usually C or Java. All projects gave an overview of compiler theory, the phases of the compiler, and the interactions between components. All projects described language design criteria and argued for their language design according to the criteria described. All projects gave a semantic description of (parts) of their language, most using structural operational semantics accompanied by a structured description in English. There has, however, been the tendency after 2009 that the semantic descriptions are less formal than previously.

Courses. As seen in Figure 2, there are four courses in this semester: two SE courses with general knowledge and two PE courses (shaded gray) that are directly related to the development projects. The courses provide conceptual foundations for technologies, methodologies, modeling, and concepts relating to programming languages, compilers, syntax, and semantics. The 3-ECTS-point PE course on Syntax and Semantics introduces formal languages, finite state and push-down automata, grammars, and structural operational semantics. The 3-ECTS-point PE course on Languages and Compilers introduces programming paradigms, syntax, parsing, type checking, and code generation and a number of compiler-compiler tools. The 3-ECTS-point SE course on Computer and Network Architecture introduces the basic machine architecture of von Neumann machines and introduces electronic circuits, microprogramming, and machine programming. The course also introduces network protocols, physical networks, and the OSI layered model. The 3-ECTS-point SE course on Principles of Concurrency and Operating Systems introduces fundamental concepts of processes, concurrency, parallelism, synchronization, and intraprocess communication. It also introduces basic concepts from operating systems such as multiprogramming and security, I/O, and virtual memory.

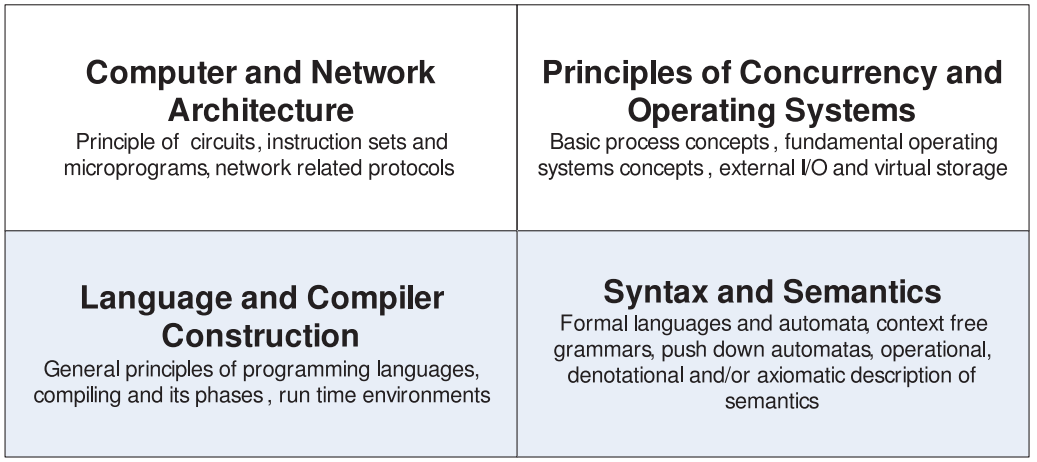

Fig. 2. Programming language technology: the fourth semester (SW4) courses.

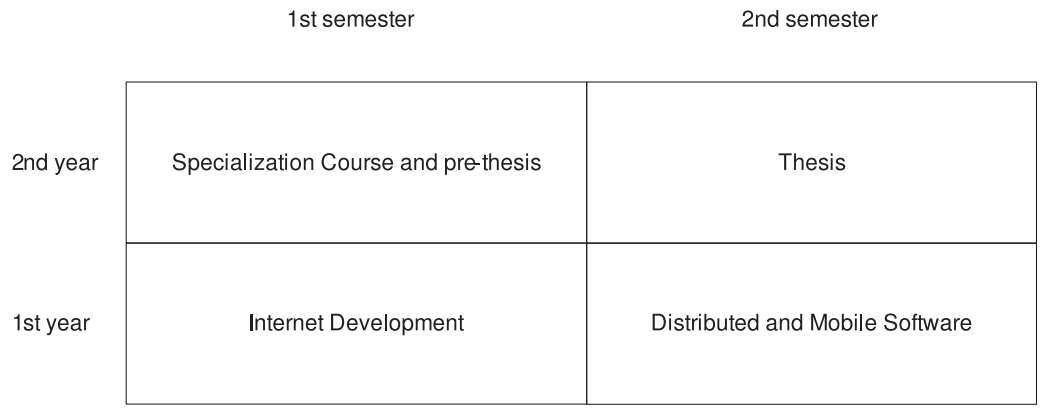

Fig. 3. Software engineering master's program at Aalborg University.

3.2. Software Engineering Master's Program

The software engineering master's program builds upon a bachelor's program in software engineering and has a nominal duration of four semesters (120 ECTS points). This is depicted in Figure 3. Further details are summarized in Appendix C. There are two main aims of the software engineering master's program at Aalborg University. The first is to broaden the skills of software engineering graduates mainly toward building more complex systems than they are used to after obtaining a bachelor's degree. This is achieved in the first two semesters. These should also serve as additional input for students to decide about their final research specialization. The second aim is to specialize through research based on the students' choice. This is achieved in the last two semesters of the master's program.

The first semester focuses on Internet development. It offers four course modules and one project module. The project module focuses on developing a web application, an agent, or a service, which has to be scalable and have an appropriate interface. The course modules are Internet Technologies, Web Engineering, Agent Technology, and Software Engineering Management. This semester is described in more detail in Section 3.2.1.

The second semester in the software engineering master's programme features mobile and distributed applications. It offers four course modules and one project module. The project module is a natural extension of web technologies with presentation options and computing realized not only on desktop, server side, or web browser platforms but also on mobile devices and technologies. The semester features courses such as Mobile Systems Technologies, which include programming techniques and languages for mobile platforms; Software Innovation; Models and Tools for Parallelism; and Advanced Algorithms.

During the third and fourth semesters of the master's program at Aalborg University, it is possible to specialize in various areas when choosing one of the specialization topics related to software engineering. Specialization is further possible in *programming,* where there are currently a number of problems offered to be solved by master's students in their research project in their final year. This includes, for example, new declarative or markup languages, combining object-oriented languages with declarative ones, or programming rich Internet applications. Students may choose from the topics offered within *intelligent web and information systems,* which are typically projects solving problems in the areas of adaptation and personalization strategies on the web, adaptive middleware for service integration, or web engineering concepts and methods. *Human computer interaction and usability evaluation* is another area where students can specialize within software engineering. This includes usability methods and studies advanced user interface design concepts in different software applications. *Software process improvement and project management* is another possible topic for specialization. *Database management* is the area where students focus mostly on business intelligence or location-aware services and connected data management issues as another specialization. *Mobile and embedded systems* specialization offers students the possibility to specialize in formal methods, test and verification, and concurrency for such applications. *Machine intelligence* provides students with an opportunity to embed machine-learning and graphical inference models into their software application in their specialization.

The specialization year is split into two semesters. The third semester of the master's program has one project module and one course module. The project module is devoted to a chosen problem within an area described previous to deepen students' knowledge in that area and perform a preliminary search for a hypothesis with relevant experiments and methodologies. The course module is devoted to studies of research papers within the particular area of specialization.

The fourth semester is completely devoted to master's thesis development and research in the chosen topic(s).

3.2.1. Internet Development: The First Semester (SW7). To make the study program more concrete, we will in the following look at the contents of the seventh semester (the first semester of the master's program).

Motivation. The subject of this semester is Internet development with the main focus on web application engineering projects. There are two main motivations for having a web development semester in the software engineering master's program:

—*Labor market*—Many software applications are developed or are now being ported to the web, so there is a market and a need for software engineers with web engineering skills.
—*Complexity*—Web applications or web systems are complex, distributed systems for a diverse audience that bring new challenges to building and maintaining them.

The main outcomes to be achieved at the competence level are that the students are able to

—demonstrate knowledge and understanding of Internet, Internet technologies, and Internet services, and
—demonstrate skills in development of an Internet application, agent, or service.

Projects and Results. It seems that the market and trends in what is built for customers have been the main drivers of student interests in choosing a problem to be solved in the web development project. The topics of the projects have ranged from web shops and enterprise resource planning systems a few years ago to, more recently, intelligent homes controlled over Internet of Things middleware. No matter which of the topics the students chose, the projects provided a good foundation for building their skills in the following:

—*Data preservation and processing*—different options for storing data in a relational database, XML technologies and their use, and so on
—*Business logic in a broader sense*—describing and representing computation even in a distributed fashion in various languages and platforms (Java, enterprise Java beans, .Net, web services, middleware, security and privacy, brokers, etc.)
—*Presentation*—designing and implementing advanced user interfaces following established HCI principles and employing and experimenting with new technologies (Silverlight, rich Internet applications, Google Web toolkit, Java server faces, etc.)
—*Team coordination*—developing advanced applications in an organized manner following modern methodologies (document centric or agile)

As examples we include the students' own descriptions of their projects:

Easy Clocking—A System for Automatically Clocking In and Out Employees. "Easy Clocking is a system capable of automatically clocking in and out employees based on presence at workstations. Embedded devices are configured and software is developed to detect the presence of butchers at workstations using a localisation technology. We analyse and compare a number of different localisation technologies from which we choose Bluetooth. Its applicability is evaluated by conducting experiments. A Web application is developed in order to view registered location information and to solve conflicts, such as if the butcher is detected at multiple workstations at the same time. Work has been done with emphasis on usability and flexibility of the Web application, making Easy clocking effective to work with and allows for customisation according to the needs of the company deploying it, respectively" [Frost et al. 2009].

Duelco—Company Web Shop Integration. "This report documents the development of the new Web shop for the company Duelco. The system is based on a 5-tier architecture, and is designed with emphasis on flexibility and usability. The system is based on an analysis of Duelcos current system and their requirements. Regarding flexibility, the tier interfaces are designed using XML Schema, and all communication is done using XML. Furthermore the project focuses on the use of XML and related technologies. Although the use of XML caused some problems, overall it helped the development. Despite software license limitations, the final system ended up as a working prototype thanks to the loosely coupled 5-tier architecture, which meant that only two tiers had to be changed to get a working system" [Andersen et al. 2005].

Analysis of Projects. We have analysed 16 projects out of those that have been completed in the web development semester. Six were new web shops or integration between web shops to achieve a business advantage. Two were web applications related to retail customers in different ways than web shops. Three were focusing on games or media. One project was an editor extension for a content management system. Finally, four projects focused on mobile applications or the Internet of Things. The projects

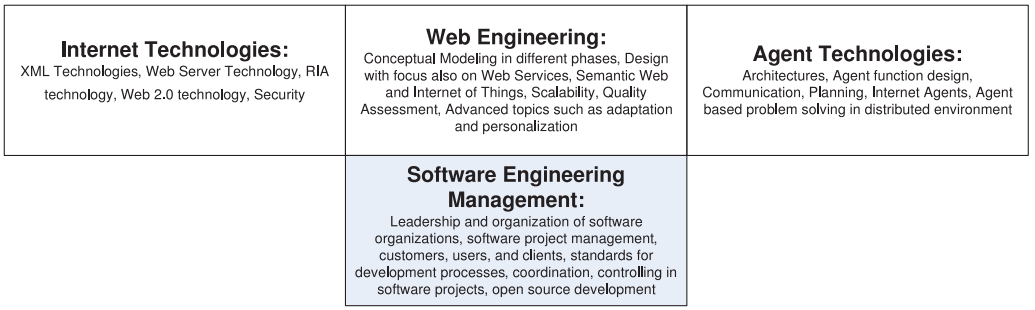

Fig. 4. Internet development: The first semester (SW7).

mostly used .Net technology (11 projects). Four projects used Java-related technology, two projects used Ruby or Google Web Toolkit technology, and one project used PhP technology. Regarding the methodology used for the project implementation, the majority of the groups chose an agile methodology based on XP or SCRUM or tailored their own methodology with elements of agile and traditional methodologies (11 projects). Seven projects followed a rather traditional document-centric methodology.

Furthermore, project-based learning has also enabled settings for various studies. Students of two consecutive instances of the semester were asked to follow the ADRIA [Dolog and Stage 2007] methodology to study understandability, applicability, and suitability of the methodology for various tasks. Another example of research in that semester was studies of various programming languages for different aspects of the developed systems and algorithms used for different computational problems. The students also most often followed user interface design evaluation based on established HCI principles. They usually used the HCI lab to perform their evaluations, involving real users. The semester projects also featured a number of university-industry collaborations. The focus in these projects was mostly some advanced applications of new technologies that a company wanted to explore in their types of projects, looking at feasibility, consequences of introducing new technology, and which features it supports.

Courses. Figure 4 depicts the lecture modules provided for students in that semester. There are three courses that are directly related to the development projects. They provide conceptual foundations for technologies, methodologies, modeling, concepts, and agents on the web. Software Engineering Management is shaded because it is a general advanced engineering course that broadens the student's knowledge beyond the project management area. The three project-related courses were especially updated quite frequently according to the wishes of the students and their project topics as well as changes in the respective fields of study. The latest additions to the curriculum were the web engineering and agent technologies courses. The motivation for the first course was to introduce web-specific conceptual, modeling, and methodological issues into the students' competencies. Besides other things, students are introduced to service-oriented architectures and business process modeling. The slot on advanced topics in the course provides flexibility for new directions in the area that might in the future include, for example, cloud computing and other new developments. The course was evidently needed according to student evaluations from previous instances of the semester. The lecture slot was previously provided for more application-oriented topics such as ERP systems, especially due to the large market share of software companies in the local area and also due to the problems previously chosen in student projects. Instead of the Agent Technologies course, students previously had an opportunity to attend a programming paradigms course, supporting their experimentation with rule-based, functional, and

aspect-oriented languages. However, the programming paradigms course was moved to an earlier semester. Similarly, students previously experimented with usability evaluation and user interface design methods. Therefore, the semester offered a course on design and use of user interfaces to provide the foundation for building web applications. This course was also moved to an earlier semester and replaced with an advanced software engineering management course.

4. ASSESSING THE STUDENTS AND THE STUDY PROGRAMS

In this section, we first discuss how students are assessed. Then we report on the official assessment and quality control process of the study program. These two subsections will hopefully convince the reader that we follow best practices at Aalborg University. However, these subsections say very little about what is gained through the PBL-based curriculum. One could of course undertake a comparative study with a non-PBL-based curriculum; such studies have been undertaken for other subject areas [Vernon and Blake 1993; Strobel and van Barneveld 2009]. However, for curriculum designers contemplating introducing PBL in their own curriculum, such comparative studies say little about what is gained. We attempt to expose what is really gained by analyzing the program according to Bloom's taxonomy [Bloom 1956] and compare it with the expected skills and competencies for an engineer passing a general software engineering 4-year program with an additional 4 years of experience as defined in the IEEE Software Engineering Body of Knowledge (SWEBOK) [Abran et al. 2004]. We also compare with the Graduate Software Engineering 2009 Curriculum Guidelines for Graduate Degree Programmes in Software Engineering (GSwE2009) [Pyster 2009].

4.1. Assessing the Students

As mentioned, in the old Aalborg model, most semesters consisted of two SE courses, two PE courses, and one project. The SE courses were rather traditional in structure and usually consisted of 15 sessions with 2-hour lectures and 2-hour exercises. The assessment of students in SE courses also usually followed a rather traditional approach with either a 4-hour written exam or a 20-minute oral exam. The assessment was done by two people; the person responsible for the course and one other person, either another member of the scientific staff (denoted internal examiner) or a person from the Danish board of external examiners (denoted external examiner). According to Danish rules and regulations, approximately 50% of exams are with internal examiners and 50% are with external examiners. Each course had a description of the learning objectives for that course and students were graded using the Danish grading scale, the 13-scale [Grafisk 2006; Wikipedia 2015a], until the change in 2007 to the 7-scale [Grafisk 2006; Wikipedia 2015a], which is aligned with the European ECTS grading scale [Ministeriet for Brn Ministeriet for Brn; Wikipedia 2015b] and comparable to the A-F grading scale. Up until 2010, there were few examples of variations on course examinations; for example Nørmark et al. [2008] report on the use of mini projects in the assessment of courses in introductory programming and the use of peer assessment in coursework [Hüttel and Nørmark 2012].

All software engineering student projects, from the first project during the first month of study to the final bachelor's or master's project, were evaluated through an oral exam with at least two examiners: the project supervisor, or supervisors, and an internal examiner or an external examiner from the Danish board of examiners in computer science. The project exam also covered the examination of the PE courses. Until 2005, the project exam took the form of a seminar with all the group members, the supervisor(s), and the examiner present. The group of students made a presentation of their work, usually taking turns in presenting parts of the work with 6 to 10 minutes allocated per student, followed by a joint discussion where the students were asked

questions related to the initiating problem of the project, the methods, and theories applied in the solution and the conclusions of the project. The project supervisor acted as moderator, and both examiners kept notes on who answered what and who made which contributions. The examiners could direct questions directly to an individual student, but in general the exam was an open discussion where each group member could pitch in with answers. Often the discussion was truly open and from time to time led to areas where even the examiners might not know all the answers. Our experience with this type of exam was that, although a 4- to 6-hour session was rather tough on concentration, it was usually very enjoyable as the discussion would challenge both students and examiners. Deep discussions were possible as the duration of a project exam would, as mentioned, be several hours, formally 45 minutes per student and the 1-hour project presentation, with a maximum of 6 hours in total. At the end of the exam session, the examiners graded the students, based on the extent to which the individual student fulfilled the project's learning objectives based on how the student answered, the extent to which the student participated in the discussion, and the extent to which the student could actively account for the analysis of the problem, the choice of methods, and the reasons for the choices made in the formulation and solution of the problem. Each student was graded individually, and it was not uncommon that students in the same project group received different grades. The formal description of the Aalborg University project exam guidelines can be found in AAU [2012], and several papers investigate the model [Hodges 2004; Kolmos and Holgaard 2007; Sandell and Welch 2004]. It is worth noticing that the Danish 13-grading scale was based directly on Bloom's taxonomy and thus is well aligned with the PBL model, whereas the 7-grading scale is not (directly) aligned with Bloom's taxonomy.

In 2005, the government passed a law forbidding any examination in groups even with individual grades, prescribing an individual examination. The law was changed again in 2011 with the change of government. From 2005 to 2011, projects were still done in project groups but assessed in an individual oral examination of each student. Several papers have elaborated on this period [Kolmos and Holgaard 2009; Hermansen 2007]. Our experience was that examination of individuals became much more superficial because there was less time to go deeper into the subject and because individual examination precluded the possibility of students going deeper into a subject based on the previous answers of a fellow student. Furthermore, the examiners had to ensure that each individual was asked questions of sufficient complexity, but with greater variation in the formulation of questions, to ensure fairness, as students would exchange knowledge of questions, thus giving the students examined last an unintended and unfair advantage.

These changes in the regulatory framework touched upon a number of issues relating to organizing and examining PBL activities. Clearly there are issues such as how student groups are formed. Is this an administrative issue, which is the case at Aalborg University in the first semester, or are students allowed and encouraged to form their own groups, as is the case in all other semesters in the computer science department. When students are allowed to form their own groups, the university has no control over the composition of the groups. Sometimes the academically strongest students form groups together; sometimes academically strong students distribute themselves among all the groups in a semester. Which is preferable is hard to say, as a group of academically strong students potentially could pull each other down instead of excelling, and when academically strong students distribute themselves more evenly they have a tendency to pull everybody up, including themselves. The intricacies of group composition have been studied in Shinde and Kolmos [2011]. A somewhat related issue is how much a supervisor should control the work process of the students. Obviously the supervisor should guide the students, but should a supervisor warn a group putting too

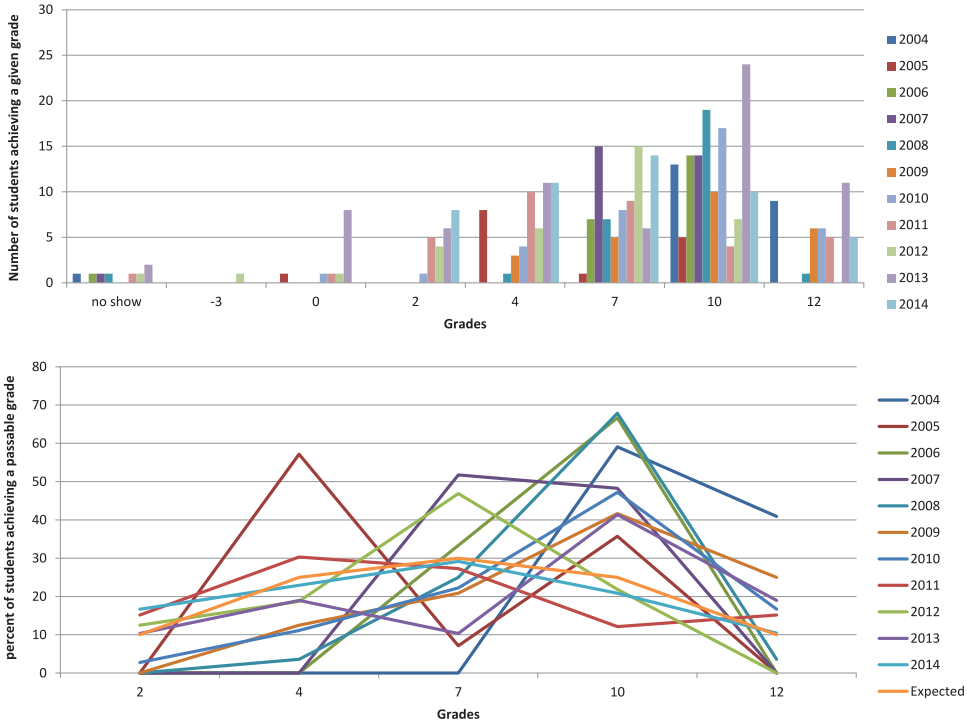

Fig. 5. Grade distribution for SW4 projects.

much effort into their project work or other learning activities? This of course begs the question: Can a university student put too much effort into his or her learning? This question is somewhat related to grading as one could argue that students should only put as much effort as needed to achieve their grading ambitions. Grading the outcome of a group-based PBL project is another issue. At some universities, grading is based only on the written report and all students in a group are given the same grade. In other places, students are asked to clearly identify the sections they were responsible for and receive individual grades based on this. In the Faculty of Engineering and Science at Aalborg University, all students in a group have joint authorship and ownership of the project report, but the project report itself is not graded. As mentioned, the project work is examined through an oral examination taking based on the written project report and the learning goals for the semester, and each student receives an individual grade.

Figure 5 shows the grade distribution of SW4 projects from 2005 to 2014. The top graph shows the actual number of students achieving a certain grade in a certain year. The bottom graph shows the percentage of students achieving a passable grade in a certain year. Note that grades are individual and, as mentioned, it is not uncommon that students from the same project group receive different grades. Grades from before 2007 have been converted from the 13-scale to the 7-scale for comparison. The 7-scale is aligned with the ECTS scale, where 10% of those who pass an exam are expected to receive the highest grade, A, corresponding to the grade 12 on the 7-scale; 25% are expected to receive the grade B corresponding to 10; 30% are expected to receive a C/7; 25% are expected to receive a D/4; and 10% are expected to receive an E/2. However, the ECTS grading system does not say anything about the expected

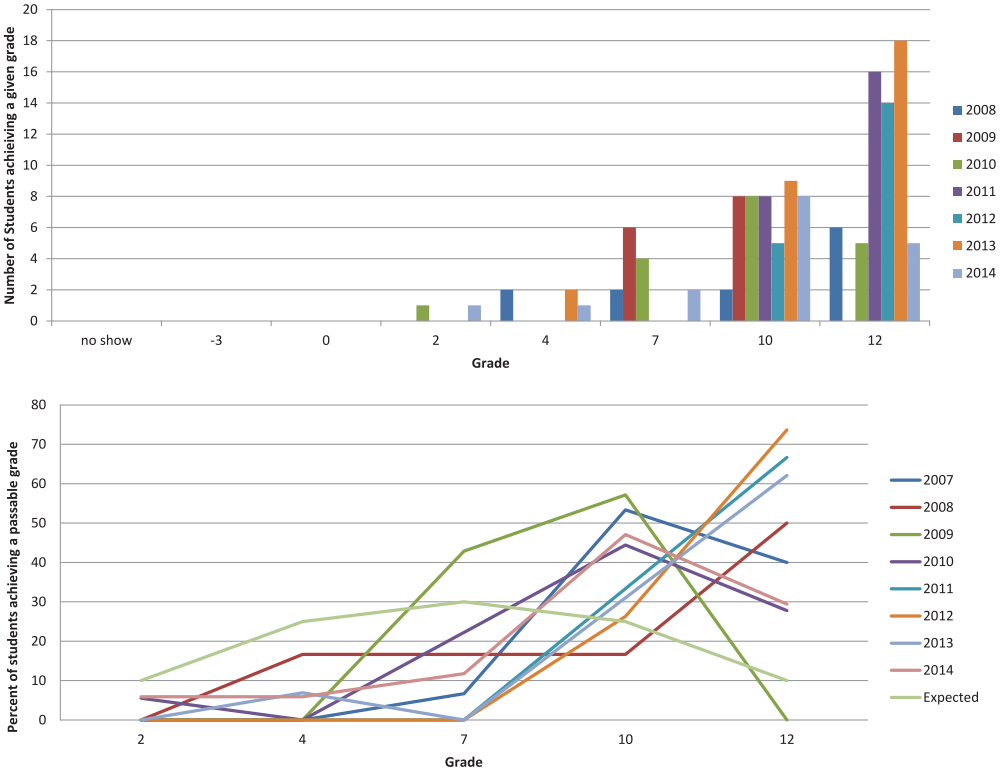

Fig. 6. Grade distribution for master's thesis projects.

failure rate. As the grades are given immediately after the oral exam, it is not possible to do any curve fitting during the exam. The top graph shows that a few students failed their SW4 project. The lower graph shows that the grade distribution does not always form a bell curve as expected (the line marked expected). Year 2011 and year 2014 are close to the expected outcome; however, year 2005 is somewhat below and the rest are above, although following a bell curve around the grade 10. Grades for SW7 projects follow a similar pattern; however, in the period from 2005 to 2014, no students failed the SW7 project exam, probably reflecting that students at this level have already achieved a high level of transferable skills in project management, teamwork, and communication of results. Figure 6 shows the grade distribution for master's thesis projects. It is very obvious that the distribution is top heavy, with most students achieving the grade 10 or 12. This probably reflects that students at Aalborg University have a lot of experience in managing and executing projects. The students also have a lot of experience in uncovering real problems and finding good solutions. It should be noted that all master's thesis projects are evaluated with external examiners, and that the learning goals for master's projects are comparable to other Danish, as well as most international, universities. Thus, it seems fair to conclude that Software Engineering master's students at Aalborg University perform above average.

4.2. Official Program Assessment Process

At Aalborg University, teaching is evaluated regularly and systematically. For each semester, the semester coordinator has designated meetings with students, supervisors, and lecturers several times during a semester to get feedback on the courses

taught as well as the progress in the projects. The final meeting also evaluates the semester in its entirety. This feedback is distributed to all involved and included in the planning of the next instance of the semester. Semester evaluations are discussed at study board meetings, and improvements/changes might be planned for the new semester.

Furthermore, all study programs are evaluated systematically and continuously. Relevant data is collected and analyzed, contributing to identifying strengths and weaknesses of individual study programs. For each study program, the quality and relevance is assessed, involving dialogue with candidates, employers, and collaborators as well as monitoring employment rates. The results are included in strategic decisions.

In addition, all new candidates are given a questionnaire about their employment situation (Dimittend undersøgelsen). Furthermore, all students finishing their bachelor's or master's degree are given a questionnaire about the study program they just completed (e.g., progression in courses, level of difficulty, workload).

Accreditation of all university study programs in Denmark was introduced in 2007 to benefit students, the labor market, and the educational institutions [ace]. Educational accreditation implies that the educational institutions must document that they fulfill a number of centrally predefined criteria regarding the need for the study programs (relevance), quality assurance, research base, and so forth. The accreditation method includes a direct assessment of whether a study program fulfills predefined quality criteria. Finally, accreditation results in an authoritative approval/nonapproval of a study program (positive/negative outcome). Accreditation of study programs is done on a rotation basis.

In 2008, both the software engineering bachelor's study program and the software engineering master's study program were accredited positive by the Accreditation Council (valid for 6 years), fulfilling 10 predefined criteria for relevance and quality. Criterion 1 required that the need for the education had to be documented, involving potential employers and the advisory board of externals. Criterion 2 covered the employability of the graduates, stressing that it had to be relevant employment. The fact that the education is research based was the focus of criterion 3, and criterion 4 confirmed that the education is based in an active research environment. The quality and strengths of this research environment had to be documented in criterion 5. Criterion 6 was about the structure of the education, for example, the progression in the courses in the various semesters. Teaching was the topic of criterion 7, covering, for example, planning of teaching, continued education of teachers, and grading of courses and projects. Quality assurance was documented for criterion 8, describing, for example, semester meetings and systematic questionnaires. Criterion 9 concerned the profile of the education, both content and title, and criterion 10 covered whether the candidates' learning outcome is as intended, based on questionnaires.

Institutional accreditation, introduced in 2013, enables institutions to establish a system that best strengthens and develops the quality and relevance of all their study programs. Institutional accreditation in general is based on a given institution's written policies, strategies, and procedures relevant to quality assurance, covering the work performed locally to strengthen and develop program quality. An institution's overall quality assurance system is assessed by the Danish Accreditation Institution. This new system is intended to simplify the accreditation process since individual study programs no longer need to be accredited if the institution is accredited positive. Aalborg University applied for institutional accreditation in 2015.

4.3. Analysis of Competence Levels According to Bloom's Taxonomy

In this section, we present an analysis of the software engineering curriculum in the context of the software engineering body of knowledge and its appendix on Bloom's

taxonomy [Bloom 1956]. The SWEBOK [Abran et al. 2004] provides a reference for a mapping to Bloom's taxonomy of skills and competencies for an engineer passing a general software engineering 4-year program with an additional 4 years of experience. We compare with the Graduate Software Engineering 2009 Curriculum Guidelines for Graduate Degree Programmes in Software Engineering (GSwE2009) [Pyster 2009] and the mapping for new graduates presented in Bourque et al. [2003], where the SWEBOK mapping of Bloom's taxonomy levels is applied to three software engineer profiles: a new graduate, a graduate with 4 years of experience, and an experienced software engineer working in a software engineering process group. We take these references as there is in general a lack of evidence from other programs to compare and explain how different curricula target certain concepts, skills, and competencies from the software engineering domain. The SWEBOK is generally known to software engineering educators, so we use it to explain our education by sampling from two semesters of the education. Although we use it in a comparative way, the intention is rather to explain than to compare, as in the SWEBOK it is not explained in detail what was the rationale behind assigning different levels to the topics from the software engineering body of knowledge. Note also that the topics here are understood as areas, in which an engineer can acquire different levels of skills and competencies. For example, a person can acquire a comprehension level in programming (in SWEBOK terminology denoted coding) of the Bloom's taxonomy by following a course in programming, applying lectured concepts in small programs in exercises, and passing the exam. On the other hand, the same person can score higher on competence level in programming (coding), for example, by choosing the right programming paradigm for a software system design of his or her own production. Similar assumptions can be made about other topics from the SWEBOK taxonomy when mapping them to Bloom's taxonomy.

In our analysis, we cover all areas of the SWEBOK (GSwE2009, respectively), focusing mainly on those topics/areas from the chosen semesters that have already been presented, but we also include topics from other semesters. The units of analysis are courses and semester projects, which by design of the activity defined in the study regulations present evidence for the analysis. Furthermore, we have analyzed all semester projects from the fourth semester (SW4), seventh semester (SW7), and 10th semester (SW10), which is the master's thesis semester.

In SW4 semester projects, in the knowledge category, students should be able to demonstrate an understanding of the formal definition of syntax and the semantics of programming languages, and demonstrate knowledge and give an overview of techniques and concepts in programming language design and implementation of compilers. Students should be able to account for the phases and connections between phases in a compiler, and they should be able to account for implementation techniques in the construction of compilers and interpreters.

In the skills category, students should be able to design a programming language, describe and define a programming language formally, and implement a programming language compiler or interpreter.

In the competence category, students should be able to evaluate the usefulness of techniques and tools for defining and implementing programming languages, and they should be able to account for how high-level programming language constructs are represented at runtime and in formal semantics.

The four courses offered on this semester are Languages and Compilers, Syntax and Semantics, Computer and Network Architecture, and Principles of Real-Time and Control Systems. The first two offer skills and competencies directly needed for the project. The latter two offer additional skills and competencies in low-level concepts in hardware and concurrency, distribution, and real-time systems behavior.

Based on the aforementioned design for this semester, we state the following hypothesis: Students design and formally describe a programming language and they build

a compiler or interpreter for this language including all phases and with appropriate interfaces between each phase. They gain knowledge and experience in understanding and selecting appropriate methods and tools for a complex software engineering task. They gain knowledge and experience in formal specification and relating it to implementation, that is, skills and competencies in software requirement, design, structure, and architecture fundamentals. Furthermore, as compilers or interpreters are large pieces of software, they gain skills and competencies in compositional software construction, that is, skills and competencies in software design strategies and methods. As the students are required to reflect on their choices, they should, by design of the curriculum, score high on Bloom's taxonomy in the process.

In SW7 semester projects, in the knowledge category, students should be able to demonstrate knowledge about and understanding of Internet, Internet technologies, and Internet services. They should be able to understand and use Internet concepts. They should be able to analyze and model requirements for Internet applications. They should be able to structure an application into a multitier architecture and to use common program patterns. Finally, the students should be able to design, realize, and test an Internet application, agent, or service.

In the second category, the skills category, the students should be able to perform systematic testing of applications to show that the application conforms to the intentions and the user needs. They should be able to perform systematic evaluation of the user interface. They should be able to argue for choices made in all development process activities and explain that requirements, architecture, and user needs are in a positive relation. They should be able to demonstrate skills in the development of an Internet application, agent, or service of high internal and external quality, where they focus on scalable architecture, a fitting user interface, and quality of service.

In the competencies category, the students should be able to develop an application that runs as an Internet application, agent, or service and solves a user problem. They should be able to describe and reflect on the applied work form and the development process.

In the four courses, they look at processes, technologies, and algorithms related to web engineering, agents, and Internet technologies complemented with software engineering management. In web engineering, the focus is on types of web applications, types of web technologies, processes, requirements analysis, design and implementation of web applications, testing with focus on performance, A/B testing, control flow, scalability, security, service-oriented integration, fault tolerance, and design patterns. In the skills category, the focus is on demonstrating knowledge of the aforementioned, performing model-based analysis and design of web applications, and applying methods from the previously noted knowledge areas to the development of web applications. In Internet technologies, the focus is on XML technologies and related programming and validation tools. The agent technologies course focuses on knowledge, skills, and competencies in theories and technologies for agents and multiagent systems.

Based on the design for this semester, we state the following hypothesis: Students build a system throughout all phases of the software engineering life cycle. Also, as required for the semester, students reflect on their work organization and process, which should allow them to score high on Bloom's taxonomy in the process. Also, in the design of the semester, there is a focus on user needs and evaluation of user interfaces. Therefore, some techniques from requirements engineering as well as evaluation/testing should be understood by students at the evaluation level of Bloom's taxonomy.

In SW10 semester projects, in the knowledge category, students should be able to demonstrate a deep knowledge and overview of a current research problem within software engineering and its possible solutions. In the competence category, the students should be able to reason scientifically about concepts and techniques, and use

and create theory in a given research area, including the identification and solution of a research problem. Furthermore, the students should be able to communicate the scientific problem, contributions to its solution, and associated concepts and techniques. In the competence category, the student should be able to analyze and contribute to a solution to a scientific problem.

We analyzed all SW4, SW7, and SW10 student projects and collected data based on the following attributes: *application domain*, *programming language*, *technology*, *methodology*, *studies*, *industry*, and *reflections*. By *application domain,* we mean the user application domain for which the application was built; *programming languages* cover which languages have been used for different components of the system; *methodology* refers to the model used for organization of work and deliverables; *industry* is used to collect information on whether there was industrial involvement of any kind in the project; and *reflections* is used to record the kinds of reflections made by the students to assess to which categories of Bloom's taxonomy the students belong and in which area. The *studies* attribute indicates whether there was any evaluation or testing for some aspects beyond traditional testing, for example, to find out about the precision of an algorithm.

We followed a number of rules to classify the level of Bloom's taxonomy for each topic of the SWEBOK (GSwE2009, respectively). We assigned the *knowledge* level for the areas where students read about them or listened to a part of a lecture about them but did not practice them. We assigned the *comprehension* level to the areas where students were able to describe them in their exercises from the courses or in their project report. The *application* level was assigned when students were able to document in their project that they have applied a technique, method, or technology in their project. The *analysis* level was assigned when we found evidence that students were able to compare technique, method, technology, and methodology in their student projects. We assigned the *synthesis* level when students were able to document that they had integrated tools, methods, techniques, or methodologies in a way not lectured in the courses. Finally, the *evaluation* level was assigned when we found evidence that students performed their own tests and reflections of technologies, tools, techniques, and methodologies designed and created by themselves.

Note that we assigned these levels even when there was only limited evidence for a particular category. We argue that this is correct under the assumption that the excellence in each category is further assessed by a grade. This is assessed by examining projects and courses and issuing individual grades. Therefore, for example, even projects where reflections on the software process chosen were at the shallow level still brought students to the level of *evaluation* but with low excellence, meaning low grade. This methodology follows the same assumptions as with assigning credit points for student learning activities. Students receive the same amount of credit points no matter which grade they score in the exam.

4.4. Results

Here we look at the differences found between taxonomies analyzed. The complete analysis can be found in Appendix A. We realized that in some areas of the software engineering curriculum the students at Aalborg University achieve a higher level of competencies than those expected for the curriculum in the SWEBOK (GSwE2009, respectively). By looking at the application domain and knowledge from those areas, we found indication that in the following areas of *software requirements fundamentals,* students at Aalborg University achieve the *analytical level* of Bloom's taxonomy *instead of the comprehension* level:

—Definition of software requirements
—Product and process requirements

—Functional and nonfunctional requirements
—Emergent properties

We explain this outcome as follows. In the requirements analysis, the students have to look at the problem domains that are not lectured and are usually outside of their area. Most of the students studied relevant literature and specific examples of existing systems under the domain. They listed a number of alternative requirements taken from those studies and documented them, usually in the form of a product backlog (a list of features). They used sketches for user interface design alternatives and they decided on certain alternatives after reflections. We argue that they are at the analytical level also in other categories, as they have been able to analyze various programming languages and technology alternatives and decide on suitable alternatives for them according to their own specified criteria, not only according to user requirements, but also according to nonfunctional requirements and emergent properties at the system level rather than at the component level.

They further achieved the *evaluation* level in *prototyping,* as this is the main tool for them to progress in their student projects.

In the categories where a customer is required, and in areas where requirements management and change management are required, the students of our program, however, scored lower than the referenced SWEBOK program. In the SWEBOK program, the authors envision graduates with 4 additional years of experience in industry, where they can acquire skills and competencies relevant for working with customers.

In *software design,* students at Aalborg University score at the *evaluation* level in *interaction and presentation design* and in *design pattern* instead of at the *application* level found in the SWEBOK example. This is because, by design, they need to acquire knowledge from lectures on those topics and they are required to evaluate this knowledge in their semester project in case of interaction design, and they need to reflect upon it in case of design patterns. We have found evidence of this in all the involved semester projects.

On the other hand, regarding *quality analysis and evaluation techniques,* we realize that the students at Aalborg University largely do not practice this. They apply testing techniques, but they do not perform higher-level analysis of quality. Therefore, they score only at the *comprehension* level and not at the *application* level.

In *software testing,* we have not found any significant differences in comparison to the SWEBOK example.

In the *software maintenance* area, students at Aalborg University score only at the *knowledge* level in the following areas: *impact analysis*, *maintainability*, *cost estimation*, *parametric models*, *experience*, and *software maintenance measurement*. This is due to the fact that the semester projects are of approximately 4 months' duration, and they are not of an evolving nature.

Regarding *software configuration management*, students at Aalborg University score at the same level as in the SWEBOK example in the areas related to tools for version management and dealing with changes. However, they score lower at the *knowledge* level in the following areas: *SCM measures and measurements*, *in-process audit of SCM*, *software configuration control board*, and *software change request process*. This is because these areas are only able to be practiced with customers and in industrial settings, or if the semester design would provide these roles within the semester project.

For *software engineering management,* we did not find any significant differences in comparison to the SWEBOK example.

Looking at the *software engineering process*, students at Aalborg University score high at the *evaluation* level in the area of *life cycle models*, as we found evidence of

reflections on decisions before the project is performed, during the project performance, and at the end of the project as part of the project reports. Also, there is usually an adapted version of a theoretical model with qualitative arguments as to why the students adapted it. On the other hand, there is a lack of quantitative evaluation and measurements in the processes during the semester. Therefore, in the subareas of *process and product measurement,* students score only at the *knowledge* level.

In *software engineering tools and methods*, students at Aalborg University score high at the *evaluation* level in the area of *software construction tools*. This is because students program larger software every semester, and they need these tools to build the programs and to collaborate as well as for version control.

In *software quality*, the students at Aalborg University score lower at the *knowledge* level in comparison to the *comprehension* or *application* levels from the SWEBOK example in the areas of *value and cost of quality*, *software process quality*, *criticality of systems*, *dependability*, and *integrity levels of software*. The students are able to practice the first two levels only in organizations with a certain maturity, while the other three relate to the application domains and they are not practiced at Aalborg University.

The Graduate Software Engineering 2009 Curriculum Guidelines for Graduate Degree Programmes in Software Engineering (GSwE2009) [Pyster 2009] introduce a few categories not in the SWEBOK, such as social, legal, and historical issues, including data confidentiality and security, surveillance, and privacy. In this category, students at Aalborg University score lower at the *knowledge* level in comparison to the recommended *comprehension* level, partly because these topics are not in the curriculum. Although some students may address such topics as part of their project work and therefore would score higher in this category, it is not mandatory and we thus note the score as low in general.

4.5. Discussion

The main advantages of the project-centered study program with respect to competencies come from the fact that most students have 10 one-semester team projects during their education. Therefore, the process management skills and competencies, especially, are acquired during learning at the university instead of students waiting until they reach employment to gain them. As an example, we look at the SW7 semester of Internet development. The semester project is solved typically by a group of five to seven students. Each student needs to contribute to be able to get a grade; thus, the project cannot be a simple solution that can be done by only one student. Typically, it also includes an external stakeholder who provides requirements for the students such as in Duelco—Company Web Shop Integration described in Section 3.2. In the same project, the students had to decide how to approach the problem and identify the driving force for planning the work. There are always two approaches, work driven by software modules and work driven by prioritized requirements. The students experience both, as well as reflect on them. Similarly, as can be seen from the analysis of the projects in Section 4.3, students for various reasons selected either a specific agile or nonagile methodology/process and explicitly discussed their decisions.

Each student project must contain reflections on the process as well as on selected methods and techniques. These reflections help students to develop better organization, better sense for how to drive the project process, and so on. The reflections can be seen in the short descriptions of the project examples in Section 3.2.1, where students reflected on difficulties with XML in Duelco Web Shop and, on the other hand, on the advantages of their architecture.

The students are better equipped with analytic skills (not only comprehension or application skills) in software configuration management, requirements management,

and requirements analysis. This is due to the larger scope of the project work in the semester project in comparison to the smaller exercises. This can, for example, be seen in the short description of the EasyClocking project, where the students reported on analysis of a number of location awareness technologies, which, of course, pose different, in this case nonfunctional, requirements. Similarly, they designed the system with customizations in mind, having reflected on the future changes in requirements.

In the area of testing, the students already in the early semesters get an opportunity to practice and combine usability evaluation and testing, unit testing, and acceptance testing techniques. Later on, they extend with formal model checking methods. Examples of testing are mentioned in short descriptions of projects by students. For example, the technologies for location awareness in EasyClocking were tested in experiments before selecting the final application. Usability testing was crucial to be able to claim that the system was designed with usability concerns in mind.

Each student project applies design patterns and adapts them to new situations or finds new ones. From this perspective, the number of student projects equip the students with an additional level of software design and construction skills. The evidence for this can be found in the student project reports from the software engineering study program. Typically, the gang-of-four patterns are practiced very often, at the latest in the Internet development semester.

Regarding conceptual comprehension, one could argue that the limited number of courses provides only partial comprehension of the curriculum topics. As we found, this is not true because the students develop additional conceptual knowledge in the projects. The projects can be seen as simple practice laboratory experiences of the lectured topics, but they also serve as an extension point for additional knowledge relevant to the particular topic of a given semester.

4.6. Limitations

The limitations of this study come from the design of the analysis, hypothesis, and methodology. First of all, there are several papers [Gluga et al. 2012a, 2012b; Thompson et al. 2008] that report on different views on how to use Bloom's taxonomy to assess software engineering and computer science curricula. We have applied one specific way of assigning Bloom's taxonomy by looking at the documentation. We have not considered the grade as a contributing factor to this assignment, only the evidence at any level of detail or argumentation. Furthermore, we have followed the assignments categorized into SWEBOK topics, even though the learning objectives of our courses and projects are stated differently. This has an advantage of comparison to the reference model, but has a disadvantage of subjective interpretations of mappings to Bloom's taxonomy and mapping from our designed learning objectives to the SWEBOK knowledge areas. In addition, the SWEBOK, as stated, provides knowledge areas, not skills or competencies. Skills and competencies can be considered orthogonal. Despite these limitations, we hope that our study contributes to understanding of project-based and problem-based learning curricula and allows a wider audience to compare their implementations of software engineering programs.

5. THE NEW STUDY PROGRAM

A revision of all the study programs was initiated in 2009 and gradually introduced in the autumn of 2010 for all first-year BSc students, including computer science and software engineering, at the Faculty of Engineering and Science at Aalborg University. For the computer science and software engineering study programs, the rollout was gradual: in 2011, the first- and second-year BSc students and the first-year MSc students started on the new structure; in 2012, the remaining semesters adapted it. Hence, the program has been fully operational since the autumn of 2012.

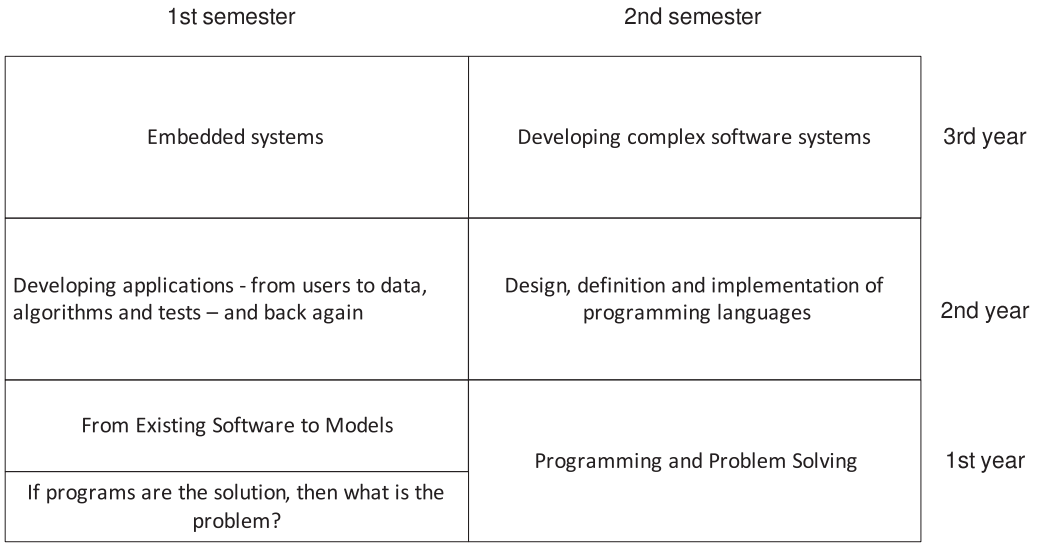

Fig. 7. Current software engineering bachelor's program at Aalborg University.

Most semesters now adhere to the so-called $3 \times 5 + 15$ model; that is, each semester has three courses where each course is credited 5 ECTS points as well as a 15-ECTS-point project. The new program is structured to be compliant with the Bologna Process and thus presents all activities in multipla of 5 ECTS points. All courses are general disciplines and must be examined separately; project-related courses are no longer part of the curriculum. This means that the students now spend only 50% of their time on projects, and the number of courses per semester has been reduced from four to three, but the ECTS count per course has increased and the overall course load has increased.

The change in structure has meant a major revision of all study programs, especially due to the reduced number of courses in each semester. In the following, we will describe the new software engineering study program both at the bachelor's level and at the master's level as of autumn 2012. Note that some students during the transition phase did not adhere to the program described here, since they had already started their study programs according to the old structure and therefore do not fit the new structure (e.g., the order of courses was slightly different or they did not have a choice between courses given their previous course program).

Most semesters still have an underlying project theme, with some progression throughout the study. Many courses will be related to the projects, but this is no longer a requirement since the concept of PE courses no longer exists. Courses are intended to provide basic curriculum knowledge as well as advanced current research topics, depending on the course and the semester. More details on the individual semesters are given later. Unless stated otherwise, the courses are 5 ECTS points and the projects are 15 ECTS points.

Figure 7 depicts the new structure of the undergraduate software engineering bachelor's study program. Further details are summarized in Appendix D. The first semester is introductory and does not adhere to the "standard" semesters since it has smaller projects. The first project is handed in after just 1 month of study, the theme being "If Programs Are the Solution, Then What Is the Problem?" (5 ECTS points). The second project has the theme "From Existing Software to Models" (10 ECTS points). During the first semester, three courses are given, namely, Linear Algebra; Problem-Based Learning in Science, Technology, and Society; and Imperative Programming.

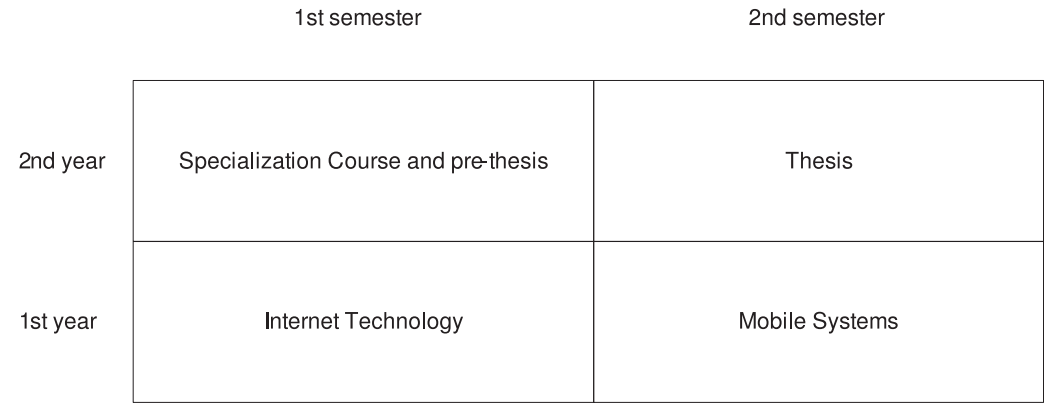

Fig. 8. Software engineering master's program at Aalborg University.

The theme of the second semester project is "Programming and Problem Solving," and the courses are Discrete Mathematics, Computer Architecture, and Object-Oriented Programming. The change in structure has also resulted in the second semester being more related to the chosen education than previously—this means that the students feel a stronger identity early on in their studies. Another consequence of this is that it can be complicated to change education after the first year; in some cases it only makes sense to do so after the first semester.

"Developing Applications - From Users to Data, Algorithms, and Tests - and Back Again" is the theme of the third semester project, where the main task is to write a large computer program. The courses are Systems Development; Design, Implementation, and Evaluation of User Interfaces; and Algorithmic and Data Structures. The underlying topic In the fourth semester is compilers, the theme being "Design, Definition, and Implementation of Programming Languages." The related courses are Syntax and Semantics, Languages and Compilers, and Principles of Operating Systems and Concurrency. The fifth semester is devoted to the theme "Embedded Systems." Four courses are on offer, and two of them are mandatory. The mandatory courses are Software Engineering and Computability and Complexity, and there is a choice between Machine Intelligence and Real-Time Systems. The theme of the project in the sixth semester is "Developing Complex Software Systems." This project is the bachelor's project. Four courses are on offer. The two mandatory courses are Databases and Theory of Science, and there is a choice between Advanced Algorithms and Semantics and Verification.

Figure 8 depicts the structure of the master's program in software engineering. Further details are summarized in Appendix E. This structure did not change from previous years; the change happened on the configuration of courses and projects. The seventh semester project theme is "Internet Technology," and four courses are offered. This time only one course is mandatory, namely, Programming Paradigms. Of the remaining three courses on Data-Intensive Systems, Web Engineering, and Agent Technologies two have to be chosen. The project theme in the eighth semester is "Mobile Systems," and the mandatory course is being Software Innovation. Of the courses on Mobile Software Technology, Advanced Programming, and Test and Verification, two have to be chosen.

The ninth semester is the so-called prespecialization semester with a slightly different structure. The project (with no specific theme) is 20 ECTS points, and there is a mandatory course on entrepreneurship. The only other course is the specialisation course (linked to the research area chosen for the ninth and 10th semesters),

and the choice depends on the topic. At the time of writing, the choice in specialization course is between Database Technology, Distributed Systems, Human-Computer Interaction, Semantics and Verification, Machine Intelligence, Systems Development, and Programming Technology. The 10th semester is devoted to the master's thesis (30 ECTS points) and the theme is a continuation of the work done in the ninth semester.

6. EXPERIENCES

Two years after introducing the new structure, it is still too early to draw general conclusions, but some preliminary conclusions can be drawn.

We have as far as possible repeated our analysis of competence levels according to Bloom's taxonomy on the small sample available. We have analyzed six SW4 projects and five SW7 projects produced by students following the first run of these semesters according to the new structure.

Of the six SW4 projects, three projects designed and implemented imperative languages and three projects designed and implemented declarative languages. Two projects developed their compilers by hand, one project used Lex/yacc, one project used SableCC, and one project used another compiler-compiler. One project generated native code, four generated high-level code, and one project implemented an interpreter. Compared to previous projects, the most recent projects gave only a rudimentary account of the general structure of a compiler. None of the projects included an analysis of the pros and cons of the various implementation techniques, and most projects were less formal in their description of semantics. The causes of the latter have been traced to a change in high school mathematics and an unfortunate assignment of supervisors without proficiency in formal semantics. The first cause is being addressed by changes in the introductory mathematics courses, and the second cause is being addressed by ensuring that supervisors assigned to this semester have the necessary skills in semantics.

Compared to previous projects, it is also noticeable that the projects in the SW4 semester now contain less material covering general theories and techniques, as would be expected as this part of the learning goals has moved from the projects to the courses and the projects have been reduced from 18 ECTS points to 15 ECTS points in workload. Therefore, it seems that in the undergraduate semesters, the projects tend to be less reflective and more focused on applying techniques and theories learned in courses. However, as the courses are now examined separately from the project, and the course goals are set to be comparable with similar courses at other universities, it is interesting to observe that the grades the students achieve in some courses, such as the Languages and Compiler course, have a significant overrepresentation of high grades. This is something that, in our opinion, undoubtedly can be ascribed to the deeper understanding students achieve through their project work.

All the SW7 projects have utilized modifications of iterative and agile development methods. All projects have also reflected on the experience in using these methods. We have, however, had several observations. In 2011, for example, the project reports largely reflected on the process and its relation to, for example, configuration management, change management, context shift, discussions in the group, and other aspects of the development process chosen. Being able to modify the process and also reflect on the experience is remarkable as it moves the student to the highest level of Bloom's taxonomy for this area. From 2012 onward, these discussions have decreased and the reflections by the students on this aspect of their project have become sparser. This decrease can be explained by the following fact. First, the project workload has been reduced from 21 ECTS points to 15 ECTS points. This means that students had to reduce their attention and scope, and it seems that in projects from 2012 and 2013, this was

the area that the students chose to not cover that deeply. In 2011, the student groups were still from the old study programs, so the students had previously experienced the larger project scopes from before the study regulations were revised. Therefore, they chose not to compromise any topic at the expense of working more than was intended for by the assigned credit points. We have seen this also from the semester evaluation reports, where students reported on this unintentional higher workload. On the other hand, student groups from 2012 onward neither reported the increased workload over assigned credit points, nor they worked for 21 credit points instead of 15. This is visible through the less deep reflections on the process aspects.

Regarding the use of programming languages in projects, five projects utilized C#, C++, or the .Net platform; four projects used the Java programming language or Javascript; three projects used PhP; and two projects employed special-purpose programming solutions such as FLEX or WebGl. Regarding other technologies and concepts, projects utilized a service-oriented architecture with the SOAP framework, Mongo document-oriented database, publish subscribe and messaging, REST, and other advanced technologies where they also reflected upon the use of them. The projects also discussed to a larger degree not only technologies and building blocks but also algorithms such as, for example, recommendation algorithms from the recommender systems area.

Regarding testing methods, there was a similar situation as with process models. Students largely applied and reflected upon two types of tests in 2011: usability and performance. From 2012 onward, the tests are done mostly as black-box tests or unit tests without much of the reflection. Only sparsely is the usability test performed or reflected upon.

The SW7 semester projects also tend to be less reflective and more focused on applying techniques and theories learned in courses, especially with respect to software processes and testing. On the other hand, the projects still present a solid analysis of technologies not even lectured and also contain deep reflections on technologies. This was found in all the projects analyzed and actually goes beyond what was found in the previous projects. This could be explained by the fact that undergraduate education now better prepares the knowledge artifacts for students in courses where these artifacts are also practiced at large in mini projects and assignments. In the higher semesters, this seems to have an effect on better understanding and better drive for the projects as well as on the scope of the projects being developed. On the other hand, while the technological parts excel, the process parts are left a little behind, which is understandable because of the reduced number of credit points assigned to the project as well as project workload.

A consequence of the new structure is that with fewer ECTS points set aside for project-related work, there is less time (or no time) for project-related ad hoc courses. Although some semesters allow the students to vary their education by a breadth or depth component, in the sense that the project theme and the elective courses do not have to match, no one did so. For example, in the SW5 semester, a student can choose a combination of a Machine Intelligence (MI) project with the Real-Time Systems (RTS) course or an RTS project with the MI course. However, all students opted for depth, either pursuing an MI project with an MI course or an RTS project with an RTS course.

Lecturers of courses that were previously project courses (PE), such as Languages and Compiler Construction (SPO) and Syntax and Semantics (SS) in the 4th semester, have had to think carefully about what material should now be covered by the course and what should be covered by the project. The SPO course now covers more tools and practical applications of tools and techniques, subjects that previously were left as part of the project work. The course also covers more subjects; for example, in addition

to code generation for virtual machines, code generation of machine code was also covered. Most Language and Compiler Construction courses include a larger exercise where students will build a small compiler for a small language or a simple language extension as part of the course [Schwartzbach 2008]. As the SPO course was kept coordinated with the SS course and aligned with the expected progress in the student projects, such an exercise was not part of the SPO course as this would have implied that the students would have to build two compilers in one semester.

With the new structure, there is the option of more variation in the delivery of courses. Previously a course was defined as 15 lectures plus exercise sessions, but now there is no "standard" course structure. Each lecturer has the option of defining a course as long as it puts enough ECTS equivalent workload on the students. As a consequence, some courses have become more PBL oriented. The deepening side of the lectured topics is now achieved by additional activities beyond lectures and simple exercise sessions, for example, through mini project extensions of the course assignments where students make a larger practical application of the lectured topics going beyond the exercises. For example, in the case of the web engineering course, the mini project is a set of assignments after each lecture where each lecture's main topic is practiced and reflected upon. Students compare XML with RDF as data formats, compare design patterns in the context of a Model View Controller and reflect upon their use in the context of an application over selected larger XML datasets from http://www.cs.washington.edu/research/xmldatasets/, or discuss design methods with regard to navigation and presentation guidelines. This further supports the projects with additional comprehension as well as applications and in many cases also with evaluations and reflections. The projects also allow students to transfer the knowledge they gained from courses in the wider and deeper context by combining it into a product that solves a particular problem. Note also that it is a requirement that the product has to work (at least to some extent). Additional activities could also be larger written assignments or student-led seminars.

After completing the first run of semesters according to the new structure, student representatives from each semester were asked to collect responses from their fellow students and present them at a gathering of academic staff in the department. During these presentations, students have commented on the importance of keeping the concept of semester themes and ensuring a relationship between (some) courses in a semester and the projects. Students concluded that it was a bad idea to make the Software Engineering course too broad in SW7, but a good idea to train students in tool use in the SPO course in SW4. In general, it was commented that former PE courses (now all courses are SE courses) should be kept aligned with the project work. These conclusions are similar to conclusions of a wider study reported in Kolmos and Holgaard [2012].

In some semesters, the courses are structured very differently in, for example, the number of lectures and amount of self-study, which causes some confusion among the students, especially that courses with very different structures all count for 5 ECTS points of effort. Here it is important to explain to the students at the start of the course how the required workload is spread across various activities, for example, lectures, assignments, self-study, and mini projects.

Another issue has been the coordination of mini projects in the SW7 and SW8 semesters, respectively. Most lecturers have decided to make mini projects (larger written assignments) a mandatory part of their course, which in some cases has resulted in conflicting deadlines as well as leaving no time for the main project work. This was an unforeseen consequence of introducing examination in all courses as part of the revised structure, and the semester planning now also covers the scheduling of assignments and mini projects to avoid conflicts wrt student workload.

It is interesting to observe that complaints about workload and confusing structure mainly came from students used to the old structure. There were noticeably fewer complaints and even some positive feedback from students who only experienced the new structure.

As a tentative general conclusion, it seems that the restructuring of the study program has enabled deep learning in some topics taught through courses by also allowing student activities to be split into other activities such as mini projects. This was achieved at the expense of having the semester projects, especially in the undergraduate semesters, but also in graduate semesters, reduced to a smaller scope. However, if properly coordinated, it seems that the new structure provides more opportunities for students to learn and practice topics that they previously omitted from their projects. This is also partly seen in the higher semesters such as SW7, where students seem to be more mature and more knowledgeable and able to reflect more deeply on the topics they have learned that far.

To remedy the reduced reflective nature of the projects, it seems that supervision of the projects and coordination of the semesters require more attention and perhaps also more time. Projects as hands-on experience, where students are co-constructing the knowledge, need to be designed such that they have a learning goal of reflections, analysis, and evaluation attached. This probably reduces the students' freedom in choosing the direction as well as the scope. It would be beneficial to upgrade the students' competence level in the areas where reflections on problems have become weaker with the new study regulations.

However, as can be seen from Figure 5, the change does not seem to have affected the grade distribution for master's thesis projects.

The new Aalborg Model was partly introduced for economic reasons. As each university presumably has its own way of allocating resources to teaching tasks, it can be hard to give general recommendations or conclusions. However, it should be noted that there are no tuition fees in Denmark and that Aalborg University receives the same financial support from the government for each student as any other university in Denmark. Full-time faculty staff, such as assistant, associate, and full professors, on average spend 50% of their time teaching, corresponding to 420 hours per semester. Other groups of employees may teach less or even more; that is, a PhD student teaches 125 hours per semester, usually as project supervisor or as teaching assistant on a course, and a full-time teaching assistant spends 840 hours teaching. Aalborg University strives to have 80% of teaching covered by full-time faculty staff and 20% covered by other employees. In the old model, 1 teaching hour was allocated per student per ECTS project work. Thus, for a group of six students, a supervisor would be allocated 108 hours. A lecturer would be allocated 70 hours per ECTS lectured course, and if more than 50 students attended a course, a further 23.3 hours for a teaching assistant were allocated. Thus, a semester with 100 students would be allocated 2,920 teaching hours. A full bachelor's and master's program with 100 students in each semester needed 30 full-time faculty staff. In the new model, where nearly all projects take up 15 ECTS points, a supervisor is allocated 90 hours for supervision of six students. The allocation scheme for lectured courses has also changed. In the new model, the allocation is 150 hours plus 1.5 hours per student per course. Thus, in a semester with 100 students, only 2,400 hours are allocated for teaching. This corresponds to saving one full-time faculty member per semester, and thus a full bachelor's and master's program with 100 students needs only 24 full-time faculty staff. These new norms are partly responsible for the greater variation in the delivery of lectured courses as it is no longer possible to give a 5-ECTS-point lectured course in the traditional format of 15 sessions with 2-hour lectures and 2-hour exercises. Instead, more self- or peer-learning activities have had to be introduced. Whether this resource model can be used in other universities

is unknown, but it seems for a while to have stabilized the university expenditure on teaching against the allocations made by the Danish government.

7. CONCLUSIONS

In this article, we have presented and compared two different configurations of a problem-based learning curriculum set up for the software engineering bachelor's and master's programs at Aalborg University. The first configuration of the program follows what has become known as "the Aalborg Model" [Barge 2010; Kolomos et al. 2004], which we refer to as the old Aalborg Model. We analyze the program wrt competence levels according to Bloom's taxonomy and compare it with the expected skills and competencies for an engineer passing a general software engineering 4-year program with an additional 4 years of experience as defined in the IEEE Software Engineering Body of Knowledge (SWEBOK) [Abran et al. 2004]. We also compare with the Graduate Software Engineering 2009 Curriculum Guidelines for Graduate Degree Programmes in Software Engineering (GSwE2009) [Pyster 2009]. We have realized that software engineering students at Aalborg University achieve higher levels of competencies in a number of areas, such as software requirements fundamentals, specification, validation, and testing. These higher competence levels are mainly achieved through the project work, which, in the old Aalborg Model, plays a central role and takes up 60% to 80% of the students' study time. In addition to their problem-solving skills, students achieve enhanced transferable skills, such as project management and improved communication, negotiation, and conflict resolution. These are well-known skills obtained through group-based problem-based learning.

When analyzing grades for master's thesis projects, it turns out that students at Aalborg University seem to perform above average. We attribute this, in part, to the fact that students at Aalborg University accumulate a lot of experience in uncovering real problems and finding good solutions, as well as plenty of experience in managing and executing projects.

The second configuration, following the new Aalborg Model, has been rolled out since 2010. In this model, project work has been reduced to taking up 50% of the time students spend on their studies, and lectured courses now take up the other 50%. We have highlighted changes made and some impact the changes have had on the students and on teaching. From our observations, it seems that reducing the workload in projects has had an impact on the scope and sometimes also on the reflections in student projects. On the other hand, it has enabled students to extend their learning activities toward the topics, skills, and competencies provided in lectured courses. It is important to keep a semester theme where the project and some course activities complement each other. As lectured courses have integrated elements of PBL (e.g., mini projects or larger assignments), more coordination between courses and projects in a semester is needed to ensure that the student workload is evenly distributed and that students are not overloaded at certain times during the semester.

The old Aalborg Model for problem-based learning has been exported to a number of universities worldwide. However, the model has often been seen as an all-or-nothing model, meaning that a complete restructuring of an existing curriculum would be needed to introduce the Aalborg Model. The new program is designed to adhere to the Bologna Process; thus, all activities are given in multipla of 5 ECTS points. This facilitates easier mobility for students wanting to take part in their education at Aalborg University or for students at Aalborg University wanting to take part in their education elsewhere. We are therefore convinced that the description of the new program could serve universities wishing to try out some of the problem-based project-based learning concepts in their study programs but not necessarily wanting to introduce the Aalborg Model in all semesters.

APPENDIX
A. COMPARISONS OF BLOOM TAXONOMY

Legend:

A: Taxonomy Level	Original classification from SWEBOK[1]
B: AAU level/SWEBOK	Comparison of AAU level to SWEBOK
C: NG Level	New Graduate Level[2]
D: GSwE2009 Level	Core Body of Knowledge from GSwE2009[3]
E: AAU/GSwE2009	Comparison of AAU level to GSwE2009

[1] Alain Abran, James W. Moore, Pierre Bourque, Robert Dupuis, and Leonard L. Tripp (Eds.). 2004. Guide to the Software Engineering Body of Knowledge 2004 Edition (SWEBOK 2004). IEEE Computer Society.

[2] Pierre Bourque, Luigi Buglione, Alain Abran, and Alain April. 2003. Bloom's taxonomy levels for three software engineer profiles. In Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on. IEEE, 123–129. doi: 10.1109/STEP.2003.6

[3] A Pyster. 2009. Graduate Software Engineering 2009 (GSwE2009) Curriculum Guidelines for Graduate Degree Programs in Software Engineering. Integrated Software and Systems Engineering Curriculum (iSSEc) series (2009). Curriculum Guidelines for Graduate Degree Programs in Software Engineering. 2009

Software Requirements

Breakdown of Topics	A	B	C	D	E
1. Software requirements fundamentals				C/AP	C/AN
Definition	C	AN			
Product and process req.	C	AN			
Functional and non-functional requirements	C	AN			
Emergent properties	C	AN			
Quantifiable requirements	C	C			
System requirements and software requirements	C	C			
2. Requirements process				C	C
Process models	C	C			
Process actors	C	C			

Software Design

Breakdown of Topics	A	B	C	D	E
1. Software design fundamentals				C/AP	C/AN
General design concepts	C	C			
Context of software design	C	C			
Software design process	C	C			
Enabling techniques	AN	AN			
2. Key issues in software design				AP	AP
Concurrency	AP	AP			
Control and handling of events	AP	AP			
Distribution of components	AP	AP			
Error and exception handling and fault tolerance	AP	AP			

Process support and management	C	C			
Process quality and improvement	C	C			
3. Initiation and Scope Definition				AP	AN
Determination and negotiation of requirements					
Feasibility analysis					
Process for requirements review/revision					
3. Requirements elicitation				AP	AP
Requirements sources	C	C			
Elicitation techniques	AP	AP			
4. Requirements analysis				AN	AN
Requirements classification	AP	AP			
Conceptual modeling	AN	AN			
Architectural design and requirements allocation	AN	AN			
Requirements negotiation	AP	C			
5. Requirements specification				AP	AP
System definition document	C	C			
System requirements specification	C	C			
Software requirements specification	AP	AP			
6. Requirements validation				AP	AP
Requirements reviews	AP	AP			
Prototyping	AP	E			
Model validation	C	C			
Acceptance tests	AP	AP			
7. Practical considerations				C/AP	C/AP
Iterative nature of requirements process	C	C			
Change management	AP	AP			
Requirements attributes	C	C			
Requirements tracing	AP	AP			

Interaction and presentation	AP	E - now AP			
Data persistence	AP	AP			
3. Software Structure and Architecture				AP/A	AN
Architectural structures and viewpoints					
(macro architectural					
Design patterns (micro architectural patterns)					
Human computer interface design					
Families of programs and frameworks					
4. Software Design Quality Analysis and Evaluation				AP	AP
Quality attributes					
Quality analysis and evaluation techniques					
Measures					
5. Software Design Notations				AP	AP
Structural descriptions (static)					
Behavioral descriptions (dynamic)					
6. Software Design Strategies and Methods				AP/A	AP/AN
General strategies					
Function-oriented (structured) design					
Object-oriented design					
Heuristic methods					
Formal methods					
Component-based design (CBD)					

Software Testing

Breakdown of Topics	**A**	**B**	**C**	**D**	**E**
1. Software testing fundamentals				AP	AP
Testing-related terminology	C	C			
Key issues	AP	AP			

Measuring requirements	AP	AP			

Software Construction

Breakdown of Topics	A	B	C	D	E
1. Software construction fundamentals				AP	AN
Minimizing complexity	AN	AN			
Anticipating change	AN	AN			
Constructing for verification	AN	AN			
Standards in construction	AP	AN			
2. Managing construction				AP	AP/E
Construction methods	C	E			
Construction planning	AP	E			
Construction measurement	AP	AP			
3. Practical considerations				AP	AP/AN/E
Construction design	AN	AN			
Construction languages	AP	E			
Coding	AN	AN			
Construction testing	AP	AP			
Construction quality	AN	AN			
Integration	AP	AP			

Software Maintanance

Breakdown of Topics	A	B	C	D	E
1. Software maintenance fundamentals				C	C
Definitions and terminology	C	C	C		
Nature of maintenance	C	C	C		
Need for maintenance	C	C	C		
Majority of maintenance costs	C	C	C		
Evolution of software	C	C	C		
Categories of maintenance	AP	AP	AP		
2. Key issues in software maintenance				AP	K
Technical					

Relationships of testing to other activities	C	C			
2. Test levels				AP	AP
The target of the tests	AP	AP			
Objectives of testing	AP	AP			
3. Test techniques				AP	AP
Based on tester's intuition and experience	AP	AP			
Specification-based	AP	AP			
Code-based	AP	AP			
Fault-based	AP	AP			
Usage-based	AP	AP			
Based on nature of application	AP	AP			
Selecting and combining techniques	AP	AP			
4. Test-related measures				AP/AN	AP/AN
Evaluation of the program under test	AN	AN			
Evaluation of the tests performed	AN	AN			
5. Test process				C/AP	AP
Management concerns	C	C			
Test activities	AP	AP			

Software Configuration Management

Breakdown of Topics	A	B	C	D	E
1. Management of the SCM process				C/AP	C/E
Organizational context for SCM	C	C			
Constraints and guidance for SCM	C	C			
Planning for SCM					
SCM organization and responsibilities	AP	C			
schedules	AP	C			
Tool selection and implementation	AP	C			
Vendor/subcontractor control	C	C			
Interface control	C	C			
Software configuration management plan	C	C			
Surveillance of software configuration management					

Limited understanding	C	K	C		
Testing	AP	K	AP		
Impact analysis	AN	K	AP		
Maintainability	AN	K	C		
Management issues		K			
Alignment with organizational issues	C	K	C		
Staffing	C	K	C		
Process issues	C	K	C		
Organizational	C	K	C		
Maintenance cost estimation					
Cost estimation	AP	K	C		
Parametric models	C	K	C		
Experience	AP	K	C		
Software maintenance measurement	AP	K	C		
3. Maintenance process				AP	K
Maintenance process models	C	K	AP		
Maintenance activities					
Unique activities	AP	K	C		
Supporting activities	AP	K	C		
4. Techniques for maintenance				AP	K
Program comprehension	AN	AN	AP		
Reengineering	C	C	C		
Reverse engineering	C	C	C		

Software Engineering Management

Breakdown of Topics	**A**	**B**	**C**	**D**	**E**
1. Initiation and scope definition			C	AP from req.	AP
Determination and negotiation of requirements	AP	AP	AP		
SCM measures and measurement	AP	E			
In-process audits of SCM	C	E			
2. Software configuration identification				AP	AP
Identifying items to be controlled					
Software configuration	AP	AP			
Software configuration items	AP	AP			
Software configuration item relationships	AP	AP			
Software versions	AP	AP			
Baseline	AP	AP			
Acquiring software configuration items	AP	AP			
Software library	C	C			
3. Software configuration control				AP	AP
Requesting, evaluating and approving software changes			AP		
Software configuration control board	AP	R	C		
Software change request process	AP	R			
Implementing software changes	AP	AP	C		
Deviations & waivers	C	C	AP		
4. Software configuration status accounting			AP		-
Software configuration status information	C	C	C		
Software configuration status reporting	AP	AP	C		
5. Software configuration auditing			C		-
Software functional configuration audit	C	C	C		
Software physical configuration audit	C	C			
In-Process audits of a software baseline	C	C	AP		
6. Software release management and delivery			C	AP	C/AP
Software building	AP	AP	AP		
Software release management	C	C	AP		

Feasibility analysis	AP	AP			
Process for requirements review/revision	C	C	AP		
2. Software project planning			C	AP	AP/E
Process planning	C	C			
Determine deliverables	AP	E	C		
Effort, schedule, and cost estimation	AP	AP	C		
Resource allocation	AP	AP	C		
Risk management	AP	AP	C	Larger	AP
Quality management	AP	AP			
Plan management	C	C			
3. Software project enactment				AP	C/AP
Implementation of plans	AP	AP			
Supplier contract management	C	C			
Implementation of measurement process	AP	C			
Monitor process	AN	C			
Control process	AP	C			
Reporting	AP	C			
4. Review and evaluation				C	C
Determining satisfaction of requirements	AP	C			
Reviewing and evaluating performance	AP	C			
5. Closure				C	C/AP
Determining closure	AP	AP			
Closure activities	AP	C			
6. Software engineering measurement					C
Establish and sustain measurement commitment	C	C			
Plan the measurement process	C	C			
Perform the measurement process	C	C			
Evaluate measurement	C	C			
7. Engineering Economics				C	C

Software Engineering Tools and Methods

Breakdown of Topics	**A**	**B**	**C**	**D**	**E**
1. Software tools					AP/AN

Software Engineering Process

Breakdown of Topics	**A**	**B**	**C**	**D**	**E**
1. Process implementation and change				C/AP	K/C
Process infrastructure		C	C		
Software engineering process group	C	K	C		
Experience factory	C	K	AP		
Activities	AP	K	C		
Models for process implementation and change	K	K	C		
Practical considerations	C	K			
2. Process definition			C	C	K/AN
Life cycle models	AP	AN	C		
Software life cycle processes	C	AP	C		
Notations for process definitions	C	K	C		
Process adaptation	C	K	C		
Automation	C	K			
3. Process assessment			C	AP	K
Process assessment models	C	K	C		
Process assessment methods	C	K			
4. Product and process measurement			AP	AP	C/AP
Software process measurement	AP	C	AP		
Software product measurement	AP	C	AP		
Size measurement	AP	AP	AP		
Structure measurement	AP	AP	AP		
Quality measurement	AP	AP	C		
Quality of measurement results	AN	AP			
Software information models		C	C		
Model building	AP	C	C		
Model implementation	AP	C			
Measurement techniques		C	C		
Analytical techniques	AP	C	C	SYS	
Benchmarking techniques	C	C		SYS	

Software requirements tools	AP	AP			
Software design tools	AP	AP			
Software construction tools	AP	AN			
Software testing tools	AP	AN			
Software maintenance tools	AP	AP			
Software engineering process tools	AP	AP			
Software quality tools	AP	AP			
Software configuration management tools	AP	AN			
Software engineering management tools	AP	AP			
Miscellaneous tool issues	AP	AP			
2. Software engineering methods					AP
Heuristic methods	AP	AP			
Formal methods and notations	C	AP			
Prototyping methods	AP	AP			
Miscellaneous method issues	C	AP			

Preparation Knowledge	**BSc Level**	**AAU Level**
Mathematics Fundamentals		
1. Discrete Structures	**AP**	AP
Functions, relations, and sets; basic logic; proof techniques; basics of counting; graphs and trees; discrete probability		
2. Propositional and Predicate Logic Propositions, operators, and truth tables, laws of logic, predicates and quantifiers, argument and inference	**AP**	AP
3. Probability and Statistics Basic probability theory, random variables and probability distributions, estimation theory, hypothesis testing, regression analysis, analysis of variance	**AP**	AP
Computing Fundamentals		
1. Programming Fundamentals	**AP**	E

Software Quality

Breakdown of Topics	**A**	**B**	**C**	**D**	**E**
1. Software quality fundamentals				AP	C/AP
Software engineering culture and ethics	AN	C	AN		
Value and costs of quality	AN	C	AP		
Quality models and characteristics		C			
Software process quality	AN	C	AP		
Software product quality	AN	C	AP		
Quality improvement	AP	C	C		
management processes				AP	C/AP
Software quality assurance	AP	AP	C		
Verification and validation	AP	AP	AP		
Reviews and audits					
Inspections	AP	AP	C		
Peer reviews	AP	AP	C		
Walkthroughs	AP	AP	AP		
Testing	AP	AP	AP		
Audits	C	C	C		
3. Practical considerations					C/AP
Application quality requirements					
Criticality of systems	C	C	C		
Dependability	C	C	C		
Integrity levels of software	C	C	C		
Defect characterization	AP	AP	C		

Overview of programming languages; virtual machines; introduction to language translation; declaration and types; abstraction mechanisms; object-oriented programming; functional programming; language translation systems; type systems; programming language semantics; programming language design		
2. Data Structures and Algorithms	**C**	AP
Basic algorithmic analysis; algorithmic strategies; fundamentals of computing algorithms; distributed algorithms		
3. Computer Architecture	**C**	AP
Digital logic and digital systems; machine level representation of data; assembly level machine organization; memory system organization and architecture; interfacing and communication; functional organization; multiprocessing and alternative architectures; performance enhancements; architecture for networks and distributed systems		
4. Operating Systems	**C**	C
Operating system overview and principles; concurrency; scheduling and dispatch; memory management; device management; security and protection; file systems; real-time and embedded systems; fault tolerance; system performance evaluation; scripting		
5. Networks and Communications	**C**	K

Software quality management techniques					
Static techniques	AP	AP	AP		
People-intensive techniques	AP	AP	AP		
Analytic techniques	AP	AP	AP		
Dynamic techniques	AP	AP	AP		
Software quality measurement	AP	AP	AP		
3. Verification and Validation (V&V)				AP	AP
Definitions of V&V		AP			

Introduction to net-centric computing; communication and networking; network security; Internet; building Web applications; network management; compression and decompression; multimedia data technologies; wireless and mobile computing		
6. Module Design and Construction	**AP**	AP
Abstraction, information hiding, interface design, procedural design, assertions, exceptions,		
Software Engineering		
1. Software Requirements	**C**	AN
Software requirements fundamentals; requirements elicitation; requirements analysis;		
3. Software Construction	**AP**	E
Software construction fundamentals; software construction practices		
4. Software Testing	**K**	AP
Software testing fundamentals; test levels; test techniques		
5. Software Maintenance	**K**	K
Software maintenance fundamentals; techniques for maintenance		
6. Software Engineering Management	**K**	AP
Software project planning; software configuration management		
7. Software Engineering Process	**K**	AP
Process definition and implementation; product and process measurement		
8. Software Quality	**K**	K
Software quality fundamentals; software quality management practices		

- System V&V and software V&V		AP			
- Independent V&V		AP			
V&V Techniques		AP			
- Testing		AP			
- Demonstrations		AP			
- Traceability		AP			
- Analysis		AP			
- Inspections		AP			
- Peer reviews		AP			
- Walkthroughs		AP			
- Audits		AP			

B. SOFTWARE ENGINEERING BACHELOR PROGRAM – OLD AALBORG MODEL

Semester	Module	ECTS	Activity Type	Evaluation Form	Internal/External
1 (SW1)	If programs are the solutions – what is the problem?	5	Project	Pass/No-Pass	Internal
	From existing software to models	14	Project	7-scale	Internal
	Programming in C	2	PE-Course	Pass/No-Pass	Through Project
	Problem based learning, technology and society	2	PE-Course	Pass/No-Pass	Through Project
	Assesment of IT systems	2	PE-Course	Pass/o-Pass	Through Project
	Mathematics 1A	5	SE-Course	7-scale	Internal
2 (SW2)	Programming and Problem solving	17	Project	7-scale	External
	Discrete mathematics	2	PE-Course	7-scale	Through Project
	Collaboration, learning and Project management	2	PE-Course	7-scale	Through Project
	Science, humanity and society	2	PE-Course	7-scale	Through Project
	Programming in Java	2	SE-Course	7-scale	Internal
	Matlab	2	SE-Course	7-scale	Internal
	Mathematics 2A	3	SE-Couse	7-scale	Internal
3 (SW3)	Object Oriented Programming	17	Project	7-scale	Internal
	Object Oriented Programming	3	PE-Course	7-scale	Through Project
	System Analysis and Design	3	PE-Course	7-scale	Through Project
	Algorithms and data Structures	5	SE-Course	7-scale	Internal
	Mini Project in Programming	1	Project	Pass/No-Pass	Internal
	Software Architectures and Databases	3	SE-Course	7-scale	Internal
4 (SW4)	Programming Language Technology	18	Project	7-scale	External
	Syntax and Semantics	3	PE-Course	7-scale	Through Project
	Languages and Compilers	3	PE-Course		Through Project
	Principles of operating systems and parallelism	3	SE-Course	7-scale	External
	Computer and Network Architecture	3	SE-Course	7-scale	Internal
5 (SW5)	Embedded Systems	18	Project	7-scale	Internal
	Real-time software	3	PE-Course	7-scale	Through Project
	Programming Paradigms	3	SE-Course	7-scale	Internal
	Design and Evaluation of User Interfaces	3	SE-Course	7-scale	Internal
	Complexity and Computability	3	SE-Course	7-scale	Internal
6 (SW6)	Application Developmen	18	Project	7-scale	External
	Test and verification of Software	3	PE-Course	7-scale	internal
	Multi project Management	1	PE-Course	7-scale	Through Project
	Software Engineering	3	SE-Course	7-scale	internal
	Database Systems	3	SE-Course	7-scale	Internal
	Professional Communication and Science of Computer Science	2	SE-Course	Pass/No-Pass	internal
Total		180			

C. SOFTWARE ENGINEERING MASTER PROGRAM – OLD AALBORG MODEL

Semester	Module	ECTS	Activity Type	Evaluation Form	Internal/External
1 (SW7)	Internet Development	18	Projects	7-scale	Internal
	Internet technologies	3	PE-Course	7-scale	Through Project
	Web Development	3	PE-Course	7-scale	Through Project
	Agent technology	3	PE-Course	7-scale	Through Project
	Software management	3	SE-Course	7-scale	Internal
2 (SW8)	Distributed and Mobile Software	18	Projects	7-scale	External
	Mobile software technology	3	PE-Course	7-scale	Through Project
	Models and tools for parallelism	3	SE-Course	7-scale	Internal
	Advanced Algorithms	3	SE-Course	7-scale	External
	Software Innovation	3	SE-Course	7-scale	Internal
3 (SW9)	Pre-specialization	27	Project	7-scale	External
	Specilization Course in Database Technology (Elective)	3	SE-Course	7-scale	External
	Specilization Course in Distributed Systems (Elective)	3	SE-Course	7-scale	External
	Specilization Course in Semantics and Verification (Elective)	3	SE-Course	7-scale	External
	Specilization Course in Machine Intelligence (Elective)	3	SE-Course	7-scale	External
	Specilization Course in systems Development (Elective)	3	SE-Course	7-scale	External
	Specilization Course in Programming technology (Elective)	3	SE-Course	7-scale	External
4 (SW10)	Master Thesis	30	Project	7-scale	External
Total		120			

D. SOFTWARE ENGINEERING BACHELOR PROGRAM – NEW AALBORG MODEL

Semester	Module	ECTS	Activity Type	Evaluation Form	Internal/External
1 (SW1)	If programs are the solutions – what is the problem?	5	Project	Pass/No-Pass	Internal
	From existing software to models	10	Project	7-scale	Internal
	Linear Algebra	5	Course	Pass/No-Pass	Internal
	Problem based learning, technology and society	5	Course	Pass/No-Pass	Internal
	Imperative Programming	5	Course	Pass/o-Pass	Internal
2 (SW2)	Programming and Problem solving	15	Project	7-scale	External
	Discrete mathematics	5	Course	7-scale	Internal
	Computer Architecture	5	Course	7-scale	Internal
	Object Oriented Programming	5	Course	7-scale	Internal
3 (SW3)	Development of applications – from users to data, algorithms and test – and back again	15	Project	7-scale	Internal
	Systems development	5	Course	7-scale	Internal
	Design and Evaluation of user interfaces	5	Course	7-scale	Internal
	Algorithms and data Structures	5	Course	7-scale	External
4 (SW4)	Design, Definition and Implementation of programming Languages	15	Project	7-scale	External
	Syntax and Semantics	5	Course	7-scale	External
	Principles of operating systems and parallelism	5	Course	7-scale	External
	Languages and Compilers	5	Course	7-scale	External
5 (SW5)	Embedded Systems	15	Project	7-scale	Internal
	Machine Intelligence (Elective)	5	Course	7-scale	Internal
	Real-time Systems (Elective)	5	Course	7-scale	Internal
	Software Engineering	5	Course	7-scale	External
	Complexity and Computability	5	Course	7-scale	External
6 (SW6)	Bachelor Project	15	Project	7-scale	External
	Advanced Algorithms (Elective)	5	Course	7-scale	internal
	Semantics and Verification (Elective)	5	Course	7-scale	internal
	Database Systems	5	Course	7-scale	External
	Science of Computer Science	5	Course	Pass/No-Pass	internal
Total		180			

E. SOFTWARE ENGINEERING MASTER PROGRAM – NEW AALBORG MODEL

Semester	Module	ECTS	Activity Type	Evaluation Form	Internal/External
1 (SW7)	Internet Technology	15	Projects	7-scale	Internal
	Data –intensive Systems (Elective)	5	Course	7-scale	Internal
	Web-engineering (Elective)	5	Course	7-scale	Internal
	Agent technology (Elective)	5	Course	7-scale	Internal
	Programming Paradigms	5	Course	7-scale	External
2 (SW8)	Mobile Systems	15	Projects	7-scale	External
	Mobile software technology (Elective)	5	Course	7-scale	Internal
	Advanced Programming (Elective)	5	Course	7-scale	Internal
	Test and Verification (Elective)	5	Course	7-scale	Internal
	Software Innovation	5	Course	7-scale	Internal
3 (SW9)	Pre-specialization	20	Project	7-scale	External
	Specilization Course in Database Technology (Elective)	5	Course	7-scale	External
	Specilization Course in Distributed Systems (Elective)	5	Course	7-scale	External
	Specilization Course in Semantics and Verification (Elective)	5	Course	7-scale	External
	Specilization Course in Machine Integigence (Elective)	5	Course	7-scale	External
	Specilization Course in ystems Development (Elective)	5	Course	7-scale	External
	Specilization Course in Programming technology (Elective)	5	Course	7-scale	External
	Entrepreneurship	5	Course	Pass/No-pass	Internal
4 (SW10)	Master Thesis	30	Project	7-scale	External
Total		120			

REFERENCES

Danmarks Akkrediteringsinstitution. 2015. http://akkr.dk/.

2012. *Project Examination. Guide for Students, Project Supervisors and Examiners at the Faculty of Engineering and Science, Aalborg University*. Retrieved from http://www.tek-nat.aau.dk/digitalAssets/80/80190_58843_guide_om_projekt211112_eng.pdf.

Alain Abran, James W. Moore, Pierre Bourque, Robert Dupuis, and Leonard L. Tripp (Eds.). 2004. *Guide to the Software Engineering Body of Knowledge 2004 Edition (SWEBOK'04)*. IEEE Computer Society.

ACM. 2008. ACM Computing Curricula. Retrieved from http://www.acm.org//education/curricula/ComputerScience2008.pdf.

Mads Schaarup Andersen, Kristian Bødker, Martin Madsen, Daniel Rødtness Poulsen, and Mads Vøge. 2005. *Duelco - Company Web Shop Integration*. Technical Report. Department of Computer Science, Aalborg University.

Scot Barge. 2010. *Principles of Problem and Project Based Learning: The Aalborg PBL Model*. Aalborg University. Retrieved from http://www.aau.dk/digitalAssets/62/62747_pbl_aalborg_modellen.pdf.

Anton Møller Bentzen, Frederik Højlund, Jesper Kjeldgaard, Mathies Grøndahl Kristensen, Jens-Kristian Nielsen, and Simon Stubben. 2009. *Imp: An Implicitly Parallel Language*. Technical Report. Department of Computer Science, Aalborg University.

Benjamin Samuel Bloom. 1956. *Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook I, Cognitive Domain / by a Committee of College and University Examiners ; Benjamin S. Bloom, Editor*. David McKay, New York. 207 pages.

Pierre Bourque, Luigi Buglione, Alain Abran, and Alain April. 2003. Bloom's taxonomy levels for three software engineer profiles. In *11th Annual International Workshop on Software Technology and Engineering Practice, 2003*. IEEE, 123–129.

John Cowan. 1998. *On Becoming an Innovative University Teacher*. Buckingham: The Society for Research into Higher Education and Open University Press.

Mona-Lisa Dahms. 2014. Problem based learning in engineering education. In *12th Active Learning in Engineering Education Workshop*.

Peter Dolog and Jan Stage. 2007. Designing interaction spaces for rich internet applications with UML. In *International Conference on Web Engineering (LNCS) (ICWE'07)*, Piero Fraternali, Luciano Baresi, and Geert-Jan Houben (Eds.), Vol. 4607. Springer Verlag, Como, Italy, 358–363.

Nickolas Falkner, Raja Sooriamurthi, and Zbigniew Michalewicz. 2010. Puzzle-based learning for engineering and computer science. *Computer* 99 (2010), 1. DOI: http://dx.doi.org/10.1109/MC.2009.417

Richard M. Felder, Gary N. Felder, and E. Jacquelin Dietz. 1998. A longitudinal study of engineering student performance and retention. V. Comparisons with traditionally taught students. *Journal of Engineering Education* 87, 4 (1998), 469–480. http://www4.ncsu.edu/unity/lockers/users/f/felder/public/Papers/long5.html.

Morten Friis, Anders Schjødt, Lars Ole Christensen, Bo Lind Jensen, Martin Skou, Christian Skalborg Dietz, and Rasmus Thorlsund Jensen. 2006. *BINGLE (Beginners INtermediate Declarative Graphical Language)*. Technical Report. Department of Computer Science, Aalborg University.

Christian Frost, Casper Svenning Jensen, and Kasper Søe Luckow. 2009. *Easy Clocking - A System for Automatically Clocking In and Out Employees*. Technical Report. Department of Computer Science, Aalborg University.

Richard Gluga, Judy Kay, Raymond Lister, Sabina Kleitman, and Tim Lever. 2012a. Coming to terms with Bloom: An online tutorial for teachers of programming fundamentals. In *Proceedings of the 14th Australasian Computing Education Conference - Volume 123 (ACE'12)*. Australian Computer Society, Darlinghurst, Australia, Australia, 147–156. http://dl.acm.org/citation.cfm?id=2483716.2483734.

Richard Gluga, Judy Kay, Raymond Lister, Sabina Kleitman, and Tim Lever. 2012b. Over-confidence and confusion in using Bloom for programming fundamentals assessment. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*. ACM, New York, NY, 147–152. DOI: http://dx.doi.org/10.1145/2157136.2157181

Schultz Grafisk. 2006. Bekendtgørelse om karakterskala og anden bedømmelse ved universitetsuddannelser (karakterbekendtgørelsen).

Mads Hermansen. 2007. En illustrativ case. *Dansk Universitetspdagogisk Tidsskrift* 4 (2007), s 55–57.

Linda C. Hodges. 2004. Group exams in science courses. *New Directions for Teaching and Learning* 2004, 100 (2004), 89–93. DOI: http://dx.doi.org/10.1002/tl.175

Hans Hüttel and Kurt Nørmark. 2012. *Experiences with Web-Based Peer Assessment of Coursework*. Vol. 2. SciTePress, 113–116.

Judy Kay, Michael Barg, Alan Fekete, Tony Greening, Owen Hollands, Jeffrey H. Kingston, and Kate Crawford. 2000. Problem-based learning for foundation computer science courses. *Computer Science Education* 10, 2 (2000), 109–128. DOI: http://dx.doi.org/10.1076/0899-3408(200008)10:2;1-C;FT109

Finn Kjærsdam. 2012. *ECIU - Aalborg University*. Retrieved from http://eciu.web.ua.pt/page.asp?pg=18.

Anette Kolmos and Erik de Graaff. 2014. *Problem-Based and Project-Based Learning in Engineering Education: Merging Models*. Cambridge University Press, 141–161.

Anette Kolmos and Jette Egelund Holgaard. 2007. Alignment of PBL and assessment. *Journal of Engineering Education - Washington* 96, 4 (2007), 1–9.

Anette Kolmos and Jette Egelund Holgaard. 2009. Gruppebaseret eller individuel projekteksamen. *DUT, Dansk Universitetspdagogisk Tidsskrift* 4, 7 (2009), 33–43.

Anette Kolmos and Jette Egelund Holgaard. 2012. *Evaluering af ændringerne i PBL modellen på TEKNAT AAU*.

Anette Kolomos, Flemming K. Fink, and Lone Krogh. 2004. *The Aalborg PBL Model – Progress, Diversity and Challenges*. Aalborg University Press.

Undervisning og Ligestilling Ministeriet for Brn. *Karakterskala - 7-Trins-Skalaen*. Retrieved from https://www.ug.dk/uddannelser/artikleromuddannelser/karakterskala-7-trins-skalaen.

Christina Grann Myrdal, Anette Kolmos, and Jette Egelund Holgaard. 2011. *The New Aalborg PBL Model*. Aalborg Universitetsforlag, 726–738.

Kurt Nørmark, Lone Leth Thomsen, and Kristian Torp. 2008. *Mini Project Programming Exams*. Springer, 228–242. DOI: http://dx.doi.org/10.1007/978-3-540-77934-6_18

Michael J. O'Grady. 2012. Practical problem-based learning in computing education. *Transactions on Computer Education* 12, 3, Article 10 (July 2012), 16 pages. DOI: http://dx.doi.org/10.1145/2275597.2275599

Art Pyster. 2009. Graduate software engineering 2009 (GSwE2009) curriculum guidelines for graduate degree programs in software engineering. In *Integrated Software and Systems Engineering Curriculum (iSSEc) Series*.

Karin L. Sandell and Lonnie Welch. 2004. Two examples of group exams from communication and engineering. *New Directions for Teaching and Learning* 2004, 100 (2004), 101–105. DOI: http://dx.doi.org/10.1002/tl.177

Michael I. Schwartzbach. 2008. Design choices in a compiler course or how to make undergraduates love formal notation. In *Compiler Construction*. Springer, 1–15.

Vikas Shinde and Anette Kolmos. 2011. *Students' Experience of Aalborg PBL Model: A Case Study*. Aalborg University, 197–204.

Henrik Sørensen, Troels Skakfjord Magnussen, Martin Lejsgaard Myrup, Martin Skouboe Pedersen, Kasper Guldbrand, and Rasmus Hummersgaard. 2008. *A Scripting Language for Mobile Applications*. Technical Report. Department of Computer Science, Aalborg University.

Ole Michael Spaten and Anna Imer. 2011. University Magazine about PBL Model. Retrieved from http://www.e-pages.dk/aalborguniversitet/82/25.

Johannes Strobel and Angela van Barneveld. 2009. When is PBL more effective? A meta-synthesis of meta-analyses comparing PBL to conventional classrooms. *Interdisciplinary Journal of Problem-Based Learning* 3, 1 (2009), 4.

Errol Thompson, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins. 2008. Bloom's taxonomy for CS assessment. In *Proceedings of the 10th Conference on Australasian Computing Education - Volume 78 (ACE'08)*. Australian Computer Society, Darlinghurst, Australia, Australia, 155–161.

Berlin Trendence Institute. 2011. Study on Trends in Education. Retrieved from http://www.ingenioer.aau.dk/Godt+studiemilj%C3%B8/stor+tilfredshed/.

European Union. 2010. Bologna Agreement. Retrieved from http://ec.europa.eu/education/higher-education/bologna_en.htm.

David T. Vernon and Robert L. Blake. 1993. Does problem-based learning work? A meta-analysis of evaluative research. *Academic Medicine* 68, 7 (1993), 550–63.

Wikipedia. 2015a. *Academic Grading in Denmark*. Retrieved from http://en.wikipedia.org/wiki/Academic_grading_ in_Denmark.

Wikipedia. 2015b. *ECTS Grading Scale*. Retrieved from http://en.wikipedia.org/wiki/ECTS_grading_scale.

Received March 2013; revised October 2015; accepted November 2015