Making a Year-Long Software Engineering Project Agile

Leon Sterling\textsuperscript{1\!*} and Antonio Lopez-Lorca\textsuperscript{1}

School of Computing and Information Systems, The University of Melbourne
leonss@unimelb.edu.au
antonio.lopez@unimelb.edu.au

\textbf{Abstract.} We describe the transformation of a year-long software engineering project subject from being primarily document-driven to being run with agile processes. The revised subject is currently being offered in a masters course at an Australian research university. The subject gives students great industry experience while meeting course and accreditation requirements. We claim that the agile approach better fits the needs of a modern software engineering capstone project.

\textbf{Keywords:} agile, software engineering, teaching, capstone project, year-long project

1 Introduction

It has long been recognised that it is important for software engineering degrees to include project experience with students developing software for clients with real needs [8,3,2]. Consequently, the curriculum for software engineering degrees usually contains a year-long software project subject, preferably being built for industrial clients with inexact but achievable requirements. Indeed, from 2002 onwards, software engineering degrees courses jointly accredited by Engineers Australia and the Australian Computer Society were required to have meaningful year-long project(s). The Engineers Australia Guidelines are given in Appendix A.

The year-long project subject serves two primary purposes. The first purpose is the capstone experience of allowing students to demonstrate what they have learned in subjects throughout their degree. The second purpose is giving students practical experience so that they are employment ready.

In 1996, Engineers Australia first accredited an Australian software engineering degree, at the University of Melbourne. The University of Melbourne software engineering degree included two year-long project subjects, in third year with smaller teams, and in fourth year with larger teams. The size for the smaller teams ranged from 4-6 students, while larger teams ranged from 9 to 17

\textsuperscript{*} Emeritus Professor, Centre Design Innovation, Swinburne University of Technology
students depending on the number of students enrolled and the availability of the projects.

Undergoing two year-long projects gave students invaluable experience, especially if the students were involved in peer mentoring [3]. The first author who was involved in the projects from 1995 to 2007 often received feedback of the effectiveness of the software engineering degree in producing employment-ready graduates. According to a member of the University of Melbourne industry advisory board, the project experience was arguably more useful than being trained on-site [9]. A summary of key features of the Melbourne software engineering project teaching is presented in [5].

Given the nature of the software engineering discipline from the 1970s till the early 2000s, the emphases in the project subjects were on producing documents and adhering to management plans and quality standards. The need to follow standards has been argued by many, including [4]. However the field of software engineering has evolved over the past fifteen years. Along with rapid changes of technology and the emergence of new platforms such as smart phones and tablets, there has been an increase of the use of agile processes in industry. The Agile Manifesto developed in 2001 has gradually taken hold and agile practices are much more expected these days. In the words of [12], *Both waterfall and iterative approaches are giving ground to much lighter, delivery-focused methods based on the principles the Agile Manifesto describes.*

In 2016, partly triggered by a change of teaching personnel, there was an opportunity to revamp the year-long software engineering project subject at the University of Melbourne to make the subject align better with agile processes. It was also in response to claims that university courses tend to ignore developments in industry. This paper describes the changes that were made to the year-long project over 2016 and 2017. We discuss what worked well and what was less successful. Overall, the subject is working well and we advocate the agile approach for teaching.

The paper is organised as follows. Section 2 discusses the document-centric approach that was superseded. Section 3 discusses an overview of central agile processes that were essential to incorporate. Section 4 discusses other features of the course that worked well with the changes to agile, but are not essential in a project subject adopting an agile approach. Section 5 makes some observations, while Section 6 concludes.

### 2 A document-driven approach

A challenge for teaching software engineering has always been to give students an appreciation of the importance of processes and standards that are demanded by industry. It is tempting for students that can code to think that working on a larger project or with more people is just a matter of *more of the same* of the way they go about coding assignments in their university courses. An essential of a software engineering degree, in contrast to a computer science degree, is to
explain engineering standards, quality, systems thinking and engineering trade-offs.

As the first university courses in software engineering appeared from the 1980s and onwards, they adopted the industrial approach at the time. They taught IEEE standards among others. Typically quality was used as an overriding concern.

The software engineering project subjects at the University of Melbourne were typical in their approach to documents, standards and quality. The first task students were required to do once they had been formed into teams and given a project was to develop a SQAP or Software Quality Assurance Plan. Students were required to follow the standards and processes they defined. At various stages throughout the year, the student groups were audited as to whether they were following their defined processes. Students struggled at first, but learned over the year what worked and what did not. It was an advantage to go through the process twice, which was possible to accommodate in a four year engineering degree.

Once procedures had been defined, students were required to produce a sequence of documents. A Software Requirements Specification (SRS) was first developed to capture the system requirements. Then students were required to produce two design documents – a high level Software Architecture Design Document (SADD) and a software detailed design document (SDD). In the second half of the year students developed the code and tested the code according to the test plan. Assessment for the subject was based around an assessment of these documents. The overall heuristic was that students were assessed 75% on process and 25% on product.

Going through the process of producing the documents taught the students some valuable lessons. Some excellent projects were produced. However typically the projects were not deployed. There were several problems. One was that the students spent too much time on the initial documents and not enough time on the coding, and very little time on testing and deployment. Rarely if ever was the software polished enough to be handed over with confidence that it would be extensively used. Students also left their efforts to the end, hoping they could pull some late-nighters to finish the project - a culture of hero programmers.

Furthermore, the documents produced by the students turned out not to be valuable to the clients, in contrast to what was originally promised. Requirements changed over the project development. Students struggled to understand the problem domain and had little experience of what was essential.

In 2008, the University of Melbourne introduced the Melbourne model. The four year undergraduate software engineering degree was replaced by a two year masters degree in software engineering. The year-long project subject remained and was taught in a similar way as described above.
3 Agile changes

In discussions with industry in the 2010s, it became clear that agile processes were increasingly being adopted by industry. Companies like REA Group running the popular real estate site www.realestate.com.au distributed their IT departments throughout their organisations and took an agile approach through their practices. There was an implicit demand on universities to change their practices to support a new way of working. University departments, however, are conservative in their behaviour and are reluctant to make changes. The reluctance is brought about by heavyweight quality processes related to curriculum and academic programs and incentive structures that favour research performance over teaching currency, among other factors.

In 2016, there was an opportunity to significantly change the way software engineering project subjects were taught at the University of Melbourne. There was a lot of knowledge and experience in the previous methods. Given the robust and stable teaching structure [5], how could it be made agile?

In consulting with industry about how to introduce agile in the classroom, the authors were typically directed to the agile manifesto and received an almost religious instruction about agile philosophy. Clearly there needed to be some discussion of the rationale behind agile methods in lectures. A two hour lecture encompassing the agile manifesto and agile methods such as scrum proved adequate. Reference material was useful – we found that the Mountain Goat site [1] run by Mike Cohn was helpful. More generally, given the nature of the subject where students were teaching themselves, theory was less critical than practice. Of course, how much needs to be covered in lectures is affected by how much agile material is covered in other subjects in the curriculum.

There are challenges in what experience students have. User stories were required to capture the voice of the customer. They were of varying quality and not in the comfort zone of most of the students, but did work after a fashion in most of the projects. Getting customers to prioritise as is expected in agile is hard due to lack of student skills and experience in communicating with clients.

In order to meet professional requirements we needed to ensure that the students are exposed to professional tools. There was a clear case to use the Atlassian tool set [2]. While the Atlassian tools were available, their use had not been mandated. While acknowledging the value in giving students some voice in deciding among alternatives, we prefer to tell students what they must do, which is better aligned with professional practice. For example students are assigned to teams by the instructors, but we allow students to make a case if they wish to shift groups, a process which works effectively.

Training in the use of the Atlassian tools is a challenge but there is a lot of material online, and we have discovered that students share knowledge among themselves. Once the tools were mandated, we could perform assessment by looking what had been posted in the tools rather than requiring the students to

---

[1] www.mountaingoatsoftware.com
make a paper submission. This worked effectively. The tools students used primarily were Confluence, JIRA and Bitbucket. Communication within the team could be done by HipChat, but many groups preferred to use Slack. Bamboo server was adopted by several teams. We note there were a few glitches in tool use, and that institutional support is essential for a successful use of the tools.

The main structural change in the subject was to organise the lectures and submissions around sprints. How long should each sprint be? Since the project subject was 25% of the course load, a 4 week sprint in the university course corresponds to a week of work which seemed like a natural division. Given that the semester length is 12 weeks longs, having three sprints of four weeks each semester makes sense. In practice the four week sprints seemed to work well.

Given that the students needed time to be formed into teams, it made sense to wait. In 2016, there were five 4-week sprints after an initial orientation of four weeks. In 2017, the initial period was more formally described as project inception and lasted 6 weeks, followed by a practice 2 week sprint. Then there were four 4-week sprints. We are planning on the same breakdown in 2018.

Many of the decisions of how best to run the subject were affected by existing logistics. It was decided that team sizes would be around 10 students per team. Direct contact hours for the subject were a two hour lecture per week, followed a one hour workshop the following day. The teaching team was a lecturer in charge and a team of supervisors who would keep oversight of the teams. From experience having a supervisor in charge of one or two teams worked, and required 3-4 hours per week of oversight per team.

We incorporated four agile ceremonies – sprint planning meeting, daily stand-up, sprint review meeting, and sprint retrospective meeting – to our sprints. Students are responsible for organising planning meetings, sprint reviews and retrospectives. Supervisors encourage students to demo their product to their clients in each sprint review meeting and they attend the retrospective meetings to help students reflect on the sprint.

A key process is regular stand-ups. We were keen to adopt them in the class. In 2016, we ran them sometimes in class and sometimes not. Scheduling a time when all ten team members would be present is problematic given the different timetables and work commitments of the students. It was decided it was best to use the two hour lecture slot for the stand-ups. This had the advantage of keeping students accountable and giving some comparison between the teams. All students were invited to attend the stand-ups from the other teams.

Stand-ups started after the project inception phase and were scheduled for thirty minutes per team. The supervisor was present to oversee, but the students needed to run the stand-ups themselves. It took time for the students to develop their own style. There were challenges due to the cross-cultural nature of the teams – two thirds of the students were Chinese students with different background experiences and communication styles than the local students. What should be discussed in sprints are blockers to the project, but until the team bonds, the stand-ups also serve as a team catch up. Initially, we just go around the group with students saying what they have done since they last met.
Each team arranges an additional one-hour weekly meeting with their supervisor where they discuss other project issues and may conduct another stand-up meeting. The in-class stand-ups always occur on a Tuesday (our allocated lecture day). It is useful to schedule the second team meeting on Fridays as this partitions the week in three fragments of similar duration. Effectively, the students report to each other face to face on Tuesday and again on Friday. Spacing the meetings in this manner allows the students time to make progress in their activities.

In Semester 2, the stand-ups are more efficient and were reduced to 15 minutes per team. Roles were assigned to team members, a key one being scrum master who was responsible for leading the stand-up. Roles could change between sprints, which added to the learning experience. Some teams availed themselves of the learning opportunity, others kept a single scrum master throughout the whole project. In general roles are difficult to assign due to a lack of student experience which we address further in Section 5.

We next discuss assessment. The students need time to practise, which meant that having four assessable sprints made sense. Rather than having students submit a document by a given deadline, the teaching team inspected what had been submitted by the students in the Atlassian tool set. By attending the stand-ups and ideally also the weekly team meetings, the supervisors had an overview of the activities and capabilities of each student.

It is a challenge in team projects to assess both the group achievement and individual contributions to that achievement. It was decided to weight the accomplishment 70% to the team project and 30% to individual contributions. The team score comprised 60% for the four sprints, broken down as 4x15% and 10% for presentations, which were required at the end of each semester. In addition the teams were required to produce a 2 minute video. The breakdown was 3% for the Semester 1 presentation, 5% for the Semester 2 presentation and 2% for the video. The 30% individual contribution was broken down as 20% individual contribution to the team, 5x1% reflective reports on industry guest lectures, and 5% final individual reflection including the year-long project lessons learned. The individual contribution might be revised for the 2018 offering to reduce the individual component and give more weight to communication skills.

Another feature of agile processes is the focus on working code. Ideally there should be a deployment at the end of each sprint. That rarely happened, but did work well for a project in 2017 for the Royal Melbourne Hospital with a tool to help allocate nurses to wards. Various parts of the system were deployed seven times over the course of the year.

4 Innovative additions

While the main purpose of the re-design of the year-long project subject was to make things agile, it was an opportunity to inject innovative elements in the teaching. There were three main strands of innovation which we describe in turn: judicious use of guest lectures to reinforce the key messages of the subject,
improving the communication skills of the students, and using models derived from our research.

To be truly convincing to students in projects it helps to have extensive experience with industry. That is hard for career academics. A good way to expose student to real industry experience is through guest lectures. Guest lectures reinforced important messages such as the key role of requirements and communication, how to seriously do risk assessment, and what start up companies look like where many students aspire to work. In our experience, it is not hard to attract outside speakers to deliver lectures to the students.

In 2016, there were seven guest lecturers all of whom told interesting stories. Unfortunately attendance at the guest lectures was poor as there was nothing assessable about the content. In 2017 the guest lectures were described as part of professional development. Students were required to submit 5 reports from the seven guest lectures worth 5% of the total mark for the subject. Attendance improved markedly and the students gained from the reports. It was also an excellent communications exercise for the students in writing a reflective report on what the guest said, and highlighted some communication challenges for the international students.

Communications were a challenge on many levels. Students had little experience talking to clients who only had a high level view of their requirements, had not participated in teams as large as 10, and had not participated in stand-ups, or had much experience in splitting work using the appropriate tools. The more practice they could get in those skills the better, and the subject evolved to ensure many opportunities became communication opportunities. The reflective reports on the guest lectures are one example. Another example is individual reports on what the students learned over the year, particularly in regard to learning objectives the students were required to submit early in Semester 1. Individual reports are undoubtedly valuable, but the timing and guidelines for the reports are still being fine-tuned to ensure students get adequate feedback. Another example is students giving (moderated) peer feedback. Peer feedback is not something students are comfortable doing, and so is a good learning opportunity. This is another area which is a work in progress, both in how to administer the peer feedback and how to moderate the responses. There are always learning opportunities. One student commented on the attractiveness of a student, which is inappropriate and was best handled in a personal direct conversation. It is also useful to talk directly students to help overcome cross-cultural issues.

The third strand drew on the research of the authors in agent-oriented modelling [10]. Over the past ten years the elicitation methods for the models have been streamlined from those described in the book, for example in [6,17]. The high level models, which constitute the overall motivation for the project, can be considered as business briefs. Developing a business brief that had to be approved by the customer was an excellent communication exercise. The business brief is straightforward to develop. An elicitation session lasting up to an hour where stakeholders express the roles and the Do goals the functional goals of the system, Be goals the quality goals of the system, and Feel goals which take into
consideration emotional aspirations of how students should feel when interacting with the system, have proved valuable.

An example of a motivational model from one of the 2016 software engineering projects is given in Figure 1. The project was to develop a database of skills for trainee circus performers, and have the ability to compare student performance with the example performance. Developing the motivational model worked effectively for the circus staff who were unfamiliar with software engineering. The model was presented at an international conference on Performing Arts to explain the project to others.

5 Observations

The transformation of the subject has been successful. Despite some agile decisions in the subject, students have appreciated what they have learned in the subject. The students are aware of current practice and can understand what is genuine professional experience. Indeed several students have both received employment and intern or project experiences based directly on what they learned in the subject.

Students are shown the videos produced by the previous years’ teams during the first week of the semester. The videos work well to illustrate the variety of
project domains and to set expectations for the results and intended scope of student projects.

Carefully choosing the projects that are offered to student teams is critical. We have noticed that the project outcomes improved significantly from 2016 to 2017. The improvement is due, among other factors, to a better understanding of what constitutes a good project. It is good to have a variety of types of projects to highlight to students that the nature of software projects can vary widely. As the students socialise and talk to each other, they learn the realities of different projects while facing their own challenges.

Typically many teams struggle with the quality assurance and deployment aspects of the project. It is sensible to propose projects with a limited scope to allow more time for these activities that are often neglected. One of the key advantages of having real world clients is that students feel very motivated to work on their projects, as they perceive that there is somebody who really cares about the project. The downside is that having a non-engaged client tends to affect the morale of the team. Unfortunately it is not simple to predict how engaged a client will be when we source projects. Two types of projects that we tend to avoid are those with very exploratory requirements – for instance, research projects – and those with a significant artistic component – for instance, 3D modelling or game design. We have often seen good projects that are never deployed due to lack of technical support. To avoid later frustrations, now before we accept a project we negotiate with the client to ensure that the students will have technical support or at least an infrastructure to deploy the system.

Highlights from the projects that we have supervised during 2016 and 2017 give us some insight in the reality of the projects. One research-based project had a client who was very busy and did not have sufficient time for the students. The students worked based on their own interpretation of the requirements and we only managed to detect that they were working on the wrong problem at the end of the first semester. The ensuing discussion with the students to salvage the project was very uncomfortable, as the leader of the team had a very strong personality and did not respond well to reason. The project finally concluded with passable but underwhelming results.

In another project, the students detected that there were two independent components and chose to split the team into two sub-teams. The work progressed well throughout the semester. However, when the students attempted to integrate the two sub-components in the last few weeks, they realised that they had designed and implemented them without agreeing on a shared interface. The integration was not possible within the time frame of the subject and the client was delivered the two capabilities separately.

One project had the goal of creating a virtual 3D gallery on a web browser. In retrospective, the nature of this project was not an ideal match for our students. Much of the effort required by the project was 3D modelling to create the gallery. Many of the students spent most of the year modelling the building and surrounding area for the gallery. This did not represent an adequate challenge in terms of engineering, which limited the technical skills that the students could
learn. Besides the nature of the project, the team faced very significant difficulties communicating with each other. Team members focused on their own tasks and were not interested in knowing what the rest of the students were working on. Meetings were immensely frustrating, as questions from the few willing students or the supervisor were invariably met with silence. Requests done via HipChat were frequently ignored. Attempts to improve the situation had limited success and included setting up sub-teams to push responsibility for coordination on sub-team leaders. The team managed to deliver a good product to their clients, but never managed to interact well.

Not all projects were problematic, in fact, most of them went very well. In one of them, the students had to develop a database of skills for trainee circus performers (see Section 4). The students struggled during the first half of the year to work as a team. However, during the second semester one student stepped up to the challenge of being the scrum master. He was very people-focused and brought the whole team together. They started to work much more efficiently and ended up delivering a great product to their clients. The clients presented a paper on the system at a conference on performing arts [1].

One of the best projects that we have had in the subject was one where the students developed a tool to allocate nurses to beds in the intensive care unit of a large hospital. When the client approached us, they were doing the allocation manually, writing by hand the mapping between nurses and beds on whiteboards in each room of the unit. The complete process took around 90 minutes per shift (twice a day). The client was very motivated to get a successful product and involved their IT department in the project. The team worked together very well from the beginning. One of the students emerged as a natural leader who helped everyone be proactive and efficient. He interacted very well with the clients ensuring that feedback was incorporated as soon as possible. The team had a few students that were very strong technically. The team ended up deploying their product seven times throughout the second semester. They had a strong quality assurance process and the client has been using their product since August 2017. The system has saved about one hour in allocating effort each shift.

It is a challenge to try to ensure that all students in the team contribute equally. The challenge is usually not met. Indeed that is a problem with previous project subject assessment, and seems no worse with the agile approach. Ensuring a significant individual component is important and having a hurdle on the 30% of individual performance is essential if freeloading is to be avoided. The Atlassian toolset is very useful to track individual contribution of students. Tools such as Confluence or Bitbucket record the history of document edits and code commits. As supervisors assess students directly on the Atlassian toolset, it is a straightforward process to check for uneven contributions.

It is our observation that natural technical leaders emerge which have the most influence on the decisions that the student groups adopt. In that sense, the experience is similar to projects in industry. Not all the leaders that emerge are the best qualified. Ensuring that teams are working effectively is part of the
purview of the supervisor who may need to intervene if the project and/or team has major difficulties. Again that is a challenge in all project subjects.

Ideally an agile team would have knowledge of team capabilities which is essential for estimating size of work undertaken by students. In practice, the team members do not know the capabilities of their fellow team members, and they are inexperienced working in teams with 10 members. For almost all students it is a new experience. It means estimation is done poorly, and students need encouragement to undertake estimation at all. This is an area for future improvement.

The lack of a permanent agile software engineering physical space for our students is problematic. Real-world agile development teams tend to share the same physical space, which allows them to have physical Kanban walls and have daily stand-ups. Students have to compensate for this using virtual Kanban walls (e.g. in JIRA) and by making the most of their weekly meetings. One particularly good and innovative team, built their own makeshift portable Kanban wall and used it very effectively in weekly stand-ups to keep track of team activities.

Risk assessment is also not undertaken well. Admittedly most projects are not risky since they have been chosen by the teaching team to be achievable. Therefore maintaining a risk register, and other such standard activities, is a little artificial. It is still useful to do in a lightweight way.

Another problem is that students have little idea of what is involved in roles such as scrum master and customer liaison or product owner. Since the subject is a learning experience, it is advantageous to rotate roles. However role rotation probably leads to less productivity than if roles were carefully assigned to and remained with students over the year.

Almost certainly students will know little about the domain of their software project. Students need to assign time to learn about the domain. Learning how to account for the research the students undertake and to assess it is also a work in progress.

We note there has been some work to match agile methods with agent-oriented modelling. The work of [11] is noteworthy. There have also been other attempts to make software project subjects agile, however we do not have scope to do a detailed comparison.

6 Conclusions

Using agile methods is an excellent approach to conducting student projects, one more attuned to modern business practices. It has the advantages of discouraging hero programmers, with students being marked on each sprint, and there is no possibility to make up later for a poor sprint. The mark for each sprint is a powerful way of giving feedback. Over the past two years, students have improved their behaviour and output based on the feedback from sprints. Overall, the agile methods seem more useful for clients than the previous document heavy approach.
Acknowledgements

We would like to thank other members of the teaching team over the past two years; Liz Haywood, Gil Tidhar and Doc Wallace, and other software engineering academics at the University of Melbourne: Rachel Burrows, Philip Dart, Tim Miller and Shanika Karunasekera for their support. The work was partially supported by ARC Discovery project DP160104083.

References

Appendix A

The Institution of Engineers, Australia
The Australian Computer Society

JOINT BOARD IN SOFTWARE ENGINEERING
GUIDELINES FOR THE ACCREDITATION OF FOUR-YEAR
PROGRAMS LEADING TO BACHELOR OF ENGINEERING IN
SOFTWARE ENGINEERING
Draft 5 December 2002

To be read in conjunction with IEAusTs Manual for the Accreditation of Professional Engineering Programs and ACSs Guidelines for the Accreditation of Courses at the Professional Level

1. Graduates should be equipped with the knowledge and skills needed to undertake, at entry level, a disciplined, accountable and professional approach to the engineering of software including large-scale and/or complex software systems. This includes their specification, design, implementation, verification, testing, and maintenance, to satisfy a set of defined or evolving user requirements.

2. Accreditation panels will look for the following characteristics in a well-designed four-year program leading to the degree of Bachelor of Engineering in Software Engineering.

3. The program should impart a thorough knowledge base in the established areas of software engineering, including:
   - Supporting knowledge of computer science including data structures and algorithms, programming, computer architecture, operating systems, file and database systems, data communications and networks
   - Fundamental mathematical, abstraction and modelling skills
   - Requirements elicitation and analysis
   - Software design and architecture
   - Software risk analysis
   - Software implementation
   - Software testing, verification, validation and quality assurance
   - Software traceability and maintenance
   - Configuration management
   - Software engineering tools and metrics
   - Software engineering process
   - Software project management
   - Distributed systems
   - Basics of systems engineering
   - Software engineering standards.

4. Programs should include thorough exposure to the techniques of software-engineering best-practice as currently applied in the industry to large-scale software-intensive and software-dependent projects.
5. The attributes and skills developed should include the Generic Graduate Attributes embodied in IEAust’s accreditation criteria for professional engineering degree programs [a] and in the ACS Core Body of Knowledge for IT Professionals [b].

6. Graduates should be familiar with the IEEE-CS/ACM Software Engineering Code of Ethics and Professional Practice [c], which has been endorsed by the Joint Board.

7. As software projects of any magnitude are necessarily conducted in teams, particular attention should be paid to major project work including team skills and disciplined project management. At least one extended team project is expected in the program. Some leading programs have two such projects, in the third and fourth years, in addition to minor project work in earlier years. A major project extends over at least a semester and preferably the full year, and at least one should involve large teams. Assessment of both team and individual capability are important.

8. Desirably, an investigatory or research project should also be included.

9. A four-year professional software engineering program is not simply an extra year added to a three-year computer science degree, which has different objectives. A four-year program should develop in students, from the outset, a clear identity as software engineers and an intense focus on a disciplined and systematic approach to large-scale system design. Courses designed for computer-science or information-technology programs do not always provide the depth of understanding required for system design. Accreditation panels will need to be satisfied that the program as a whole provides a comprehensive, well-designed and integrated learning experience and is not simply an amalgam of courses designed primarily for other programs.

10. An introductory course or unit in first year, providing a clear perspective on the program as a whole, is strongly recommended. Case studies involving other engineers and IT professionals can be valuable.

11. The Joint Board does not regard software engineering as a branch of electrical engineering and does not espouse the view held in some traditional engineering circles that software engineering programs must include introductory circuit theory and other electrical engineering topics. The purpose is to develop capability in engineering software systems, as an objective in its own right.

12. However, software engineers must have an adequate understanding of the characteristics and limitations of the physical systems on which their software runs, and of the risks associated with physical-system limitations. This requires a substantial introduction to computer systems engineering.

13. Like all engineers, software engineers must have a reasonable awareness of the principal domains in which their products find application. These include business systems, and physical systems of an engineering nature. Programs should provide opportunity for their students to interact with a reasonable variety of such application areas, and should require students to comprehend user requirements in some depth and be able to engineer effective interfaces.
14. In relation to (13) it is recognised that in the senior (third and fourth) years, programs may offer opportunity to specialise in a particular domain of application, perhaps through a strand of elective units; or alternatively may cover a number of domains. The balance between breadth and depth is important. On the one hand, students should engage (through course units and/or project work) with at least one major application domain in sufficient depth to understand and handle the complexities typical of real-world interactions. On the other hand, an undergraduate degree should not confine its graduates to one application domain, but should impart sufficient familiarity with the major applications of software for graduates to be able to move between them without undue difficulty.

15. In accrediting programs carrying the title Bachelor of Engineering, IEAust will look for effective interaction with other engineering programs and teaching units within the university concerned, without prescribing the form this might take.

REFERENCES

b) The Core Body of Knowledge for Information Technology Professionals: ACS.
d) Guidelines for Accreditation of Courses in Universities at the Professional Level: ACS.
Author/s: Sterling, L; Lopez Lorca, A

Title: Making a Year-Long Software Engineering Project Agile

Date: 2018

Citation: Sterling, L; Lopez Lorca, A, Making a Year-Long Software Engineering Project Agile, Agile in Education and Training, 2018

Persistent Link: http://hdl.handle.net/11343/216471

File Description: Submitted version