A "Future-Proof" Postgraduate Software Engineering Programme: Maintainability Issues

J Paul Gibson and Jean-Luc Raffy
Le département Logiciels-Réseaux (LOR)
Telecom & Management SudParis (TMSP)
9 rue Charles Fourier, 91011, France
Email: gibson.paul@it-sudparis.eu

***Abstract*—We report on the development of a software engineering programme for Masters students. Maintainability of educational programmes is critical: there is a large initial investment in developing quality programmes and we must ensure that these programmes are "future proof". Consequently, we followed a traditional software engineering life-cycle process to develop a programme that would meet the current needs of industry, whilst also being easy to maintain with respect to future changes in industrial requirements. We show how the programme has gone through a number of refinement steps — where we have iterated through the life cycle of requirements engineering (with "client" industries), high-level design (establishment of a foundational educational architecture), implementation (by lecturers), testing (through establishment of evaluation and feedback mechanisms) and maintenance (throw updates to curriculum and course content). To conclude, we propose treating educational programmes as software, and demonstrate advantages in applying software engineering techniques for development and maintenance.**

***Keywords*-Teaching, Education, Curriculum, Software Engineering**

I. INTRODUCTION

Our institute is focused on educational programmes for telecommunications engineers. Software is playing an increasingly important role in telecommunications systems; and a strategic decision was taken, a number of years ago, to introduce a postgraduate (Masters) programme which specialised in software engineering. We have been motivated by the observations of Curran[3] and Parnas[11] concerning the need to distinguish computer science from software engineering, and the goal of making software engineering a true engineering discipline [16].

Despite following guidelines in the development of software engineering curriculae [17], [7], the programme has failed to attract the number of students that are required to make it feasible to run in the long-term. We have a capacity for teaching around 20 students (for each of the 2 years of the programme), but in the last four years we have not had more than 6 students in each year.

As a result, we have made continuous changes to the programme in order to attract more students. In particular, the programme has gradually become more specialised; moving from:

- a *software engineering* stream as part of a general information technology (IT) Masters, to
- a stand-alone Masters programme *software engineering for smart devices*, to
- its most recent incarnation as a more specialised Masters *software engineering and ambient intelligence*.

The changes have required much work, and during the process of evolving and maintaining our software engineering programme (in its different forms) we have identified the need for a process to aid us in improving our work, and the quality of the programme. As software engineers, we realised that much of our academic programme could be considered analagous to software and so we should be able to apply software engineering development principles, methods and techniques to the development of our programme. This paper reports on this insight and summarises the advantages of working within such an analogy. The results should be of interest to all developers of academic programmes, and not just those teaching software engineers.

This is work in progress: the ideas need further discussion and, in our opinion, the software engineering community is the best placed to be able to provide useful feedback. We hope in some way to be contributing to the "Push to Make Software Engineering Respectable" [12].

The remainder of the paper is structured as follows. In section II we motivate our work by introducing the premise that developing an educational programme is like developing software. In section III we give an overview of the problems in developing a software engineering programme and propose that one must consider the life-cycle of the programme analagous to the software life-cycle. Sections IV, V and VI examine the main life-cycle stages: requirements, design and implementation. Section VII reports on the different evaluation techniques that can be applied during the different stages in the life-cycle. Section VIII illustrates, by reviewing the changes made between three iterations, how the evaluation can feedback into the programme development aand maintainance. Section IX concludes with some remarks about future work.

 ISBN: 978-1-61208-165-6

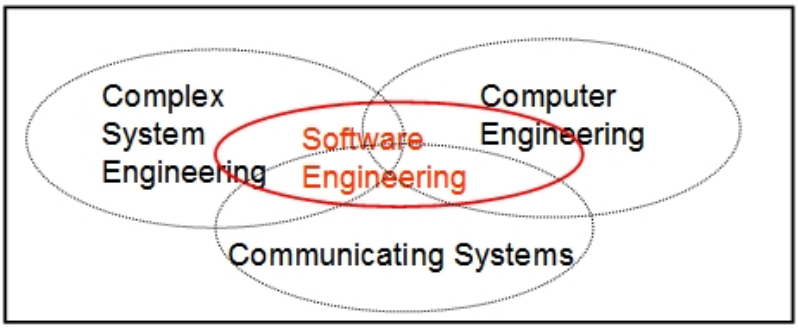

Figure 1. Software Engineering Domain of Knowledge

II. Developing an Educational Programme is like Developing Software

Most, if not all, educational programmes can be specified in a bottom-up fashion — listing a set of core components that need to be taught. Then, the traditional approach to programme development is to assign core components to programme modules and to identify dependencies between modules so that the temporal ordering of modules in the programme respects the dependency relationship. Modules are then, in general, taught and assessed independently. Irrespective of whether this is likely to produce a *good* academic programme — we would argue that it is not — it has the major disadvantage that the programme is difficult to maintain: small changes to the component requirements can have a major impact on the implementation of the programme. Further, small changes to the implementation environment (the lecturers, etc.) may necessitate major re-structuring of the programme.

Most academic programmes evolve. Unfortunately, as years pass by the links between the different versions of the programme get lost. Further, documentation of the changes made (why and how) is usually very poor. As a consequence, after a number of years there is a lack of coherence between what one is trying to teach and how it is being taught.

These type of problems are very familiar to software engineers. Managing the evolution of an academic programme, like managing a software system, has two fundamental, complementary aspects:

- a continual improvement in the understanding of the problem domain through continual analysis [13]
- an iterative life-cycle of evaluation, feedback and change [1]

Where a problem domain is well-understood and academic programmes have been well-established for a number of years then there is probably no need for a maintenance (evolution) process. However, this is clearly not the case for software engineering. The discipline has been moving so fast that many subjects that are common to recent curricula are not even mentioned in the body of knowledge from 12 years ago [2].

III. Developing a Software Education Programme: the domain and the life-cycle

A. The software engineering domain

In Figure 1 we see how the discipline of Software Engineering cannot and should not be separated from other disciplines. This figure illustrates our particular structured understanding of what we exepect our students to know about:

- Software Engineering — this is the core knowledge that all our graduates must have mastered
- Complex System Engineering — software, in general, does not exist in isolation. Most software engineering problems arise because of complex interactions in and between the software and the environment in which it exists.
- Computer Engineering — software executes on a physical machine, and we expect our students to understand how such a machine operates
- Communicating Systems — more and more software systems involve communications over different types of network. As a telecommunications school we expect our students to understand how such networks operate.

This is the view of software engineering that we believe is unlikely to change in the short to medium term (it is consistent with views on software engineering that are ten years old [17]). It is also a view that best matches our institutes' expertise (in teaching and research). Later in the paper we will introduce: the scientific and mathematical foundations upon which these four engineering disciplines are constructed, and the specific state-of-the-art techniques and tools for engineering software that have been developed out of (and interaction between) these domains.

B. The Programme Life-Cycle

As work in progress, we chose to develop our programme following the simplest, best understood, waterfall life-cycle model [15]. Such a model can be defined to different levels

 ISBN: 978-1-61208-165-6

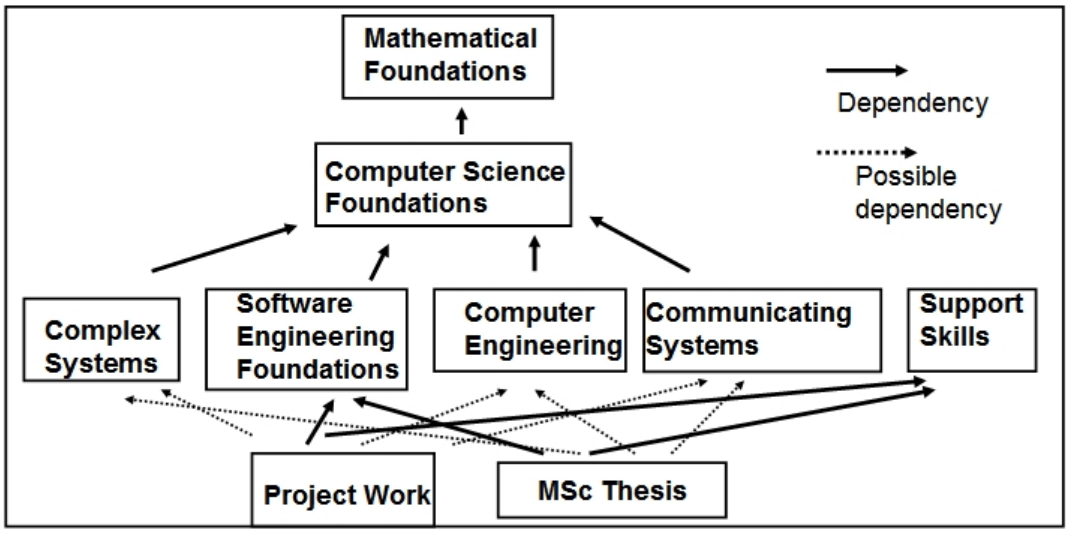

Figure 2. Architecture for Software Engineering Programme

of detail, for our initial research we chose to consider a life cycle with 3 fundamental steps — requirements, design and implementation — and feedback between each of the steps.

IV. Requirements

Our first step is to identify the requirements of our academic programme. As with complex, software systems, these requirements must attempt to meet the needs of a number of different actors:

- the students — want to improve their prospects of employment and enjoy their education;
- the lecturers — want to teach in their area of expertise, work with good quality, well-motivated students, and help promote and further their research through teaching;
- the university administration — want to attract large number of students, graduate large number of students, and minimize costs;
- the government — want their investment in universities to be coherent and worthwhile;
- industries —- want graduates that match their current needs (quantity and quality);
- research institutes — want to attract postgraduates into research careers, etc.

As programme developers we are very aware of the different compromises that exist in meeting these requirements. As such, it is critical that all interested parties are involved in the construction, evaluation and evolution of our academic programmes.

The requirements, listed above, are not specific to software engineering. We regard these in more detail in the following section - where we map specific requirements to a high-level programme design (architecture).

V. Programme Design For Meeting High-Level Requirements

In Figure 2 we represent our high-level requirements for a software engineering project:

- Our four engineering domains depend on common computer science foundations [11].
- Computer science foundations depend on mathematical foundations [5].
- All students will require support skills [14] in order to work on industrial projects and write a thesis.
- The project work [6] and thesis must demonstrate mastery of software engineering foundations, and may also depend on understanding of complex systems, computer engineering and communication systems.

The key to evolving and maintaining our programme is that these abstract components of our high-level architecture (and their interdependencies) will not change.

VI. Programme Implementation

A. Modules: The Programme Components

In Figure 3 we see how each of these abstract components is to be implemented:

- The mathematical and computer science foundations will be taught as individual modules
- The software engineering foundations will be taught as a number of inter-related modules
- The support skills — including innovation [4] — are not specific to software engineering
- The remaining modules address the 3 domains that overlap with software engineering - where we choose to address these domains through a software engineering perspective

 ISBN: 978-1-61208-165-6

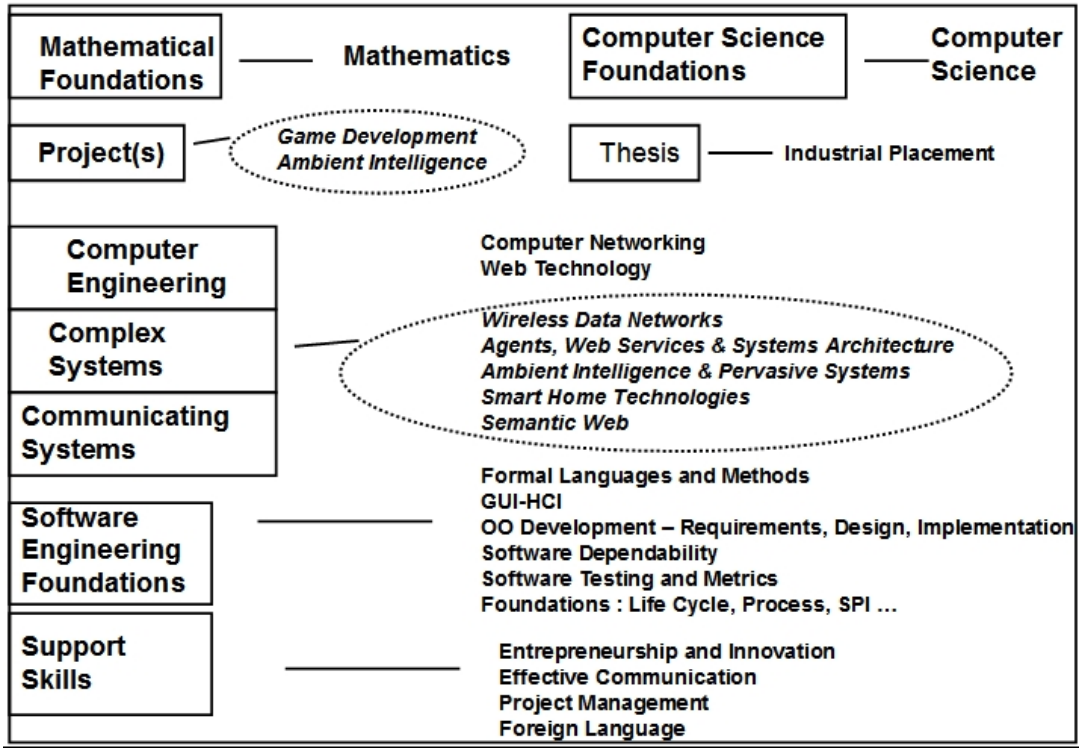

Figure 3. Implementation of Software Engineering Programme

- The project work is dependent on the the student's knowledge in these other domains
- The thesis must be written concerning an industrial placement.

From the diagram we have also circled the modules that may change over time. (The others will always be in the programme, although internal details of each of these modules may change.) The circled modules are those which are currently being taught in the latest version of the programme (*Software Engineering and Ambient Intelligence*).

B. The Approach: PBL for Software Engineering

The previous subsection examined what we are going to teach. It should also be mentioned that we have taken some decisions as to how the modules should be taught:

- Foundational mathematics and computer science will only be taught if it is used in the engineering modules. Where possible, all material in these foundational modules will be linked to the software engineering modules.
- All software engineering modules will be taught using a problem-based-learning (PBL) approach [8]. Emphasis will be on rigour and formalilty, and mathematical modelling [10]
- The PBL will draw from real-world problems taken from industries that can be expected to hire our graduates.
- Modules will be coherently connected by sharing common problems.
- The ethical side of software engineering will be emphasised and the recurrent problem of plagiarism explicitly addressed [9].

VII. Programme Evaluation

A. Accreditation

Our institute is a member of a group of schools whose Masters programmes go through an independent review (mostly by other academics) in order for them to be accredited. This accreditation is critical for attracting students as it is intended to be a good indicator of a quality programme. Further, students on accredited programmes may benefit from additional funding.

During accreditation feedback focuses on programme content. Comparisons are made with other programmes and curriculum guidelines from around the world. It is only in the current year that our programme has achieved accreditation. Many of the changes made in order to achieve accreditation were superficial in nature.

B. Industrial Feedback

A co-director for the programme is directly involved in collaboration with local industry in order to establish a *Pole de competence* for *complex system engineering*. Industries

 ISBN: 978-1-61208-165-6

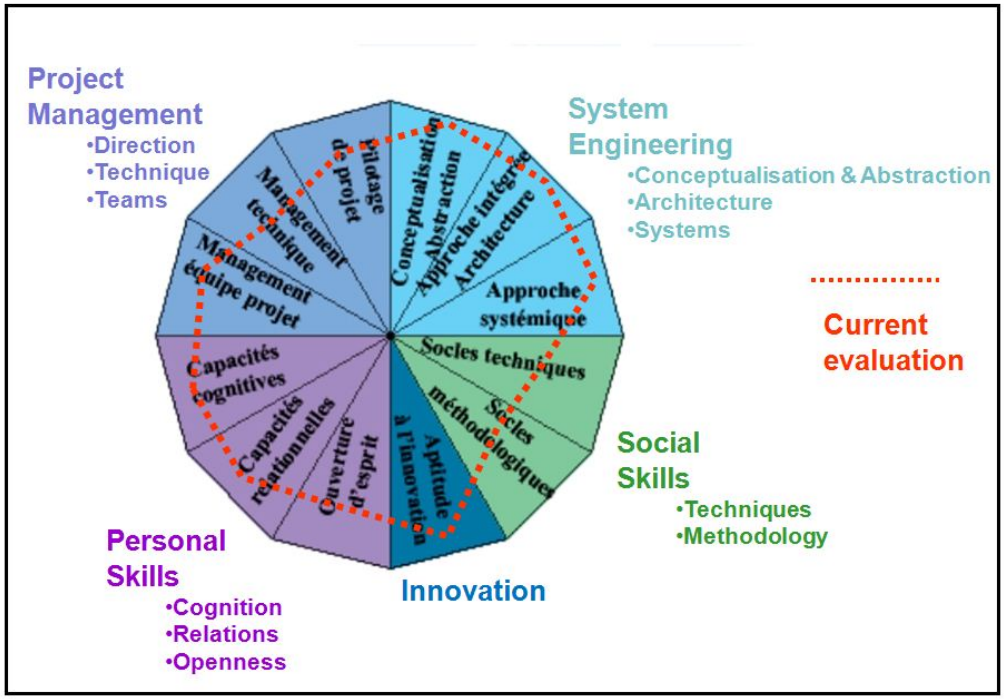

Figure 4. Industrial Evaluation Criteria

have identified a need for more (and better) software engineering students who can work in the area of complex systems (across many different industrial sectors).

In Figure 4 we see the list of competencies that these industries have identified as being core to their requirements. It should be noted that software engineering is not explicit in their criteria: in their *System Engineering* criteria they expect the engineers to be specialised in a relevant discipline (like software engineering) whilst also having generic engineering skills.

Our most recent evaluation is sketched in the diagram. This shows that we have some progress to make in all criteria, but we are weakest in *social skills*. We note that this evaluation is based on static analysis of our programme description (design and implementation).

C. Students — Quantity and Quality

Our programme will have little value if we cannot attract more students. However, we must not compromise quality for quantity. Perhaps the weakest element of our evaluation is that which we could get from students. Students are encouraged to completed feedback questionnaires concerning all aspects of the programme. With such small numbers of students, statisticaly significant analysis is not feasible. Rather, we focus on open questions and freeform discussions with students. Two main issues have arisen:

- Coherency between modules needs to be better addressed, and
- Standards of evaluation are not consistent between modules

The students suggest that increasing the number of common problems between modules will address the first issue. The second issue is more difficult to address — the students believe that the main difference is between the foundational modules (which are perceived to be difficult) and the technology modules (which are perceived to be not so difficult). These percieved differences have been validated through analysis of students' results.

We plan to keep in contact with students after they graduate. However, we have no formal procedures in place: the students rest in contact through personal communication. This requires further work, on our behalf.

VIII. Programme Evolution: Three Iterations

In 3 years, our programme has gone through 3 iterations. (In the fourth year we stabilised the programme in order to better evaluate it against the educational and industrial requirements.)

A. Software Engineering (Information Technology)

In this iteration, software engineering was taught as a specialist stream in a more general masters programme. Our initial evaluation identified weaknesses in this programme that could only be addressed by teaching a dedicated software engineering postgraduate programme:

 ISBN: 978-1-61208-165-6

- Core software engineering material was not being taught in enough detail.
- The relationship and dependencies between core material was not adequately addressed.
- Core mathematics and CS material was presented in a way that was not specific to software engineering

B. Software Engineering and Smart Devices

In this iteration we restructured our programme in order to better focus on core software engineering. For project and practical work we focused on *Smart Devices* (as this was a leading area of development in industry). Feedback from students and industry led us to prototype teaching material on developing games that would exploit the functionality of such devices. Evaluation of this programme identified weaknesses that needed to be addressed:

- We needed a clearer separation of core and non-core material.
- We needed to demonstrate the general utility of our core software engineering material by addressing more than one area in our project/practical work.
- We needed more emphasis on support skills.

This resulted in the programme as currently illustrated in Figure 3

C. Software Engineering and Ambient Intelligence

The most recent iteration has yet to be fully evaluated. We have had positive feedback from the accreditation process and from the industrial partners. However, we continue to fail to address our main weakness — there are only nine students registered for the first two years of the programme.

IX. Conclusions: remarks and future work

We have proposed treating educational programmes as software, and demonstrated advantages in applying software engineering techniques for their development and maintenance.

Current and future work involves examining re-use of material across and between programmes; and improving evaluation processes (particularly improving feedback from students).

Our major challege is not in knowing what to teach, or knowing how to teach; it is in having a reasonable number of students to teach. Perhaps we should add a "marketing" module to our curriculum?

References

[1] B Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11:14–24, August 1986.

[2] Pierre Bourque, Robert Dupuis, Alain Abran, James W. Moore, and Leonard Tripp. The guide to the software engineering body of knowledge. *IEEE Software*, 16:35–44, November 1999.

[3] W. S. Curran. Teaching software engineering in the computer science curriculum. *SIGCSE Bulletin*, 35(4):72–75, 2003.

[4] Peter J. Denning and Andrew McGettrick. Recentering computer science. *Communications of the ACM*, 48(11):15–19, 2005.

[5] Keith Devlin. Viewpoint: the real reason why software engineers need math. *Communications of the ACM*, 44:21–22, October 2001.

[6] Alan Dutson, Robert H. Todd, Spencer P. Magleby, and Carl D. Sorensen. A review of literature on teaching engineering design through project-oriented capstone courses. *Journal of Engineering Education*, 86:17–28, 1997.

[7] Gary A. Ford and Norman E. Gibbs. A master of software engineering curriculum: Recommendations from the software engineering institute. *Computer*, 22:59–71, September 1989.

[8] J. Paul Gibson. Weaving a formal methods education with problem-based learning. In T. Margaria and B. Steffen, editors, *3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science (CCIS)*, pages 460–472, Porto Sani, Greece, October 2008. Springer-Verlag, Berlin Heidelberg.

[9] J. Paul Gibson. Software reuse and plagiarism: A code of practice. In *14th ACM SIGCSE Annual Conference on Innovation and Technology in Computer Science Education(ITiCSE 2009)*, pages 55–59, Paris, France, July 2009. ACM.

[10] J. Paul Gibson, Eric Lallet, and Jean-Luc Raffy. Sculpturing Event-B models with Rodin: "holes and lumps" in teaching refinement through problem-based learning. In *From Research to Teaching Formal Methods - The B Method (TFM B'2009)*, pages 7–21, Nantes, France, 2009. APCB.

[11] David Lorge Parnas. Software engineering programmes are not computer science programmes. *Annals of Software Engineering*, 6:19–37, 1998.

[12] Gilda Pour, Martin L. Griss, and Michael J. Lutz. The push to make software engineering respectable. *IEEE Computer*, 33(5):35–43, 2000.

[13] Rubén Prieto-Díaz. Domain analysis: an introduction. *SIGSOFT Software Engineering Notes*, 15:47–54, April 1990.

[14] S.H. Pulko and S. Parikh. Teaching soft skills to engineers. *International Journal of Electrical Engineering Education*, 40(11):243–254, 2003.

[15] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[16] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7:15–24, November 1990.

[17] Mary Shaw. Software engineering education: a roadmap. In *International Conference on Software Engineering — Future of Software Engineering Track*, pages 371–380, 2000.

 ISBN: 978-1-61208-165-6