Systems Engineering for Software Engineers

Ian Sommerville
Computing Department, Lancaster University, Lancaster LA1 4YR, UK
Phone: +44-1524-593795
Fax:+44-1524-593608
E-mail: is@comp.lancs.ac.uk

Abstract

This paper describes how we have modified a software engineering stream within a computer science course to include broader concepts of systems engineering. We justify this inclusion by showing how many reported problems with large systems are not just software problems but relate to system issues such as hardware and operational processes. We describe what we mean by ‘systems engineering’ and go on to discuss the particular course structure which we have developed. We explain, in some detail, the contents of two specific systems engineering courses (Software Intensive Systems Engineering and Critical Systems Engineering) and discuss the problems and challenges we have faced in making these changes. In the Appendix, we provide details of the case studies which are used as linking themes in our courses.

1. Introduction

Software engineering gets a really bad press. All too often, we read horror stories of how software engineering projects have gone wrong with massive delays and cost overruns. The software 'crisis' which first emerged over 30 years ago is still claimed by some authors such as Pressman (Pressman 1996), who renames it as 'software's chronic affliction', to be a reality of current software development. High-profile system failures such as the Denver Airport baggage handling system (in the USA) and the London Ambulance despatching system (in the UK) have been widely publicised. Software and the poor state of software engineering has been blamed for these failures.

As an illustration of this, consider the reported problems with the Denver airport baggage handling system. This is an automated system to move baggage from aircraft to terminals which relies on software-controlled baggage carts propelled by linear induction motors. There were serious problems in the development and commissioning of this system. This delayed the opening of the new Denver airport and meant that the airport managers incurred significant costs after opening because the system was less effective than planned.

In the Scientific American of September 1994, problems with this system were discussed in an article headlined "Software's Chronic Crisis" (Gibbs 1994). The author of the article wrote:

> " *...For nine months, this Gulliver has been held captive by Lilliputians - errors in the software that controls its automated baggage system...*"

He goes on to discuss general problems with software development and engineering and illustrates these with other examples of cancelled projects which he claims were due to software failures. The general impression from this widely-read article is that the problems of the Denver airport system were exclusively software problems.

However, when we look at another account of the Denver Airport system (Swartz 1996), we see that the problems with the system were much more than software problems. They included problems of system acquisition, volatile requirements, management and hardware design. The system is immensely complex and includes:

- over 17 miles of track
- 5.5 miles of conveyors
- 4000 baggage carts (telecarts)
- 5000 electric motors
- 2, 700 photocells
- 59 bar code reader arrays
- 311 radio frequency readers
- more than 150 computers

The intention of the system was that baggage transfer would be handled automatically using a system of conveyors and baggage carts which delivered individual bags to specified destinations in the airport. The airport authorities decided to acquire a system which was based on one bag per cart (a DCV) rather than a tested system based on multi-bag carts. This was in spite of a consultancy report which stated:

"With regards to the single-bag DCV, considering the prototype state we strongly feel that it is not capable of being implemented within the project schedule"

While the system was being developed, the requirements changed radically and the software was expected to cope with the change:

" In May 1992, the airlines and the city ordered a major revision of the automated baggage system while it is under construction"

There were problems with the management of the different contractors who were responsible for developing and installing the system:

"21 October 1992: a BAE superintendent complained that another contractor was denying his crews access to the work site. Infighting continued through 1993"

The hardware design caused difficulties and the hardware did not operate correctly in some situations:

"The baggage system continued to unload bags even though they were jammed on the conveyor belt. This problem occurred because the photo eye at this location could not detect the pile of bags on the belt and hence could not signal the system to stop"

As well as all of these problems, there were also problems with the software:

"The timing between the conveyor belts and the moving telecarts was not properly synchronised causing bags to fall between the conveyor belt and the telecarts"

Therefore, we can see that the problems with this system were really much broader than simply software problems. Blaming the delays and difficulties on poor software engineering misrepresents reality. Better software engineering may have avoided some of the problems but this project was probably doomed from the outset. The system, *as a whole*, and not just the software failed to operate correctly.

A similar picture emerges in other high-profile systems failures. They are often represented in the press as being primarily software failures but, when we look at them in more detail, we see that the problems are not only software problems but are a result of more general failings in the systems engineering process.

The official report of why the London Ambulance Despatching System (CAD) failed identified other types of system problem which can arise:

" the system relied on a technical communications infrastructure that was overloaded and unable to cope easily with the demands that the CAD would place upon it particularly in a difficult communications environment such as London"

" Management clearly underestimated the difficulties involved in changing the deeply ingrained culture of the London Ambulance Service and misjudged the industrial relations climate so that staff were alienated to the changes rather than brought on board"

" the early decision to achieve full CAD implementation in one phase was misguided..."

Given these reasons for system failure why then does software get the blame? In many projects, software is delivered late and does not operate as required so, ostensibly, there is a problem with the software engineering. However, when we examine *why* the software is late, we see that the reasons are often due to underlying systems engineering problems. In order to cope with other systems problems, demands are made for software changes at a late stage in the system development. Of course, these demands for change may be made for good reasons:

- New customer requirements emerge which have to be accommodated in software because the hardware has already been designed and manufactured.
- The impact on the system on organisational processes may not have been properly analysed. When process changes become necessary, the software user interface and sometimes its structure has to be changed to accommodate these.
- Hardware incompatibilities are discovered during system integration and the software must be changed to ensure that the different hardware units can work together.
- Hardware fails to perform to its specification so additional processing by the software is required.
- Customers and system developers try to increase functionality in software rather than in hardware to avoid the costs of changing hardware whose design has been frozen.

These difficulties mean that the software specification is essentially unstable and good software engineering can only reduce the extent of the problems. Budget overruns and delays in delivery are inevitable when the developers of the software must constantly accommodate change.

It is unrealistic to think that the answer to these problems is stable software specifications. The essence of software is its malleability and the resulting flexibility is a very important reason why organisations demand more software in their computer-based systems. We must accept that other problems of systems engineering will inevitably mean that there will be demands for software changes at a late stage in the development process. Cost and schedule overruns are the price we have to pay for system flexibility.

I have deliberately gone into this at some length because I believe that the issue is a very important one for software engineering educators. Many software engineering courses such as courses in requirements engineering, object-oriented development and programming methods focus on techniques for developing software which is more amenable to change. However, it is relatively uncommon to discuss the broader systems context and to consider *why* , in practice, demands for radical software changes may be requested.

This lack of understanding of software's operational context has, I believe, two consequences:

1. Some students, especially those who are self-taught programmers, do not appreciate the need for disciplined approaches to software development. They have a programming model which focuses solely on the software and they do not understand the relationships between software and other components of large, complex systems.
2. Graduates from computer science and software engineering courses do not understand the problems faced by engineers and managers from other disciplines. This means that communication is difficult and participation by software engineers in the systems

engineering process is limited. Important decisions affecting system software may therefore be made without informed advice on the consequences for the software of these decisions.

Our experience of working with industry on systems engineering projects has convinced us that these issues are very important. While it is clearly essential to educate students in *how* to solve software problems, we believe that it is equally important to sensitise students to the reasons *why* these problems arise. We have therefore broadened the scope of our courses to include systems as well as software engineering.

In this paper, I start by discussing what is meant by systems engineering then go on to describe how we have integrated it into our computer science course. The material covered in specific systems engineering courses is described and I explain how we gained the unexpected side effect of an improved platform for discussing ethical and professional issues. Finally, I reflect on some of the difficulties which we have encountered in introducing this new course and how successful our approach has been.

The paper is a report of educational experience rather than research. We could not find any other papers in the software engineering or the systems engineering literature which covered the relationships between system engineering education and software engineering education. An informal survey of computer science courses (based on web browsing and searching) showed that systems engineering was rarely covered in computer science courses.

Our approach is, therefore, largely based on our own understanding of systems engineering problems and feedback from industry about some of the difficulties which they have in integrating computer science graduates into their development teams. We appear to be pioneers and, inevitably, we have got some things wrong. Our goal in writing this paper is, therefore, to present our case to the community and to stimulate discussion about possible ways in which software engineering education might develop in the next century.

2. What do we mean by 'systems engineering'?

The term *system* in everyday English is used when we want to refer to something as a whole rather than as a collection of parts. We may, therefore, talk about the American 'system' of government, the Communist 'system', a 'system' for teaching reading, a baggage handling 'system', a Unix 'system', an IBM 'system', and so on. Normally, we use context and our background knowledge to sort out what we really mean by 'system' in these different usages.

Because of the different ways in which the term 'system' is used, there is scope for misunderstanding about what we mean by 'systems engineering'. In essence, the type of systems which we are interested in are *socio-technical software-intensive systems.*

These are systems where some of the components are software-controlled computers and which are used by people to support some kind of business or operational process. Systems, therefore, always include computer hardware, software which may be specially designed or bought-in as off-the-shelf packages, policies and procedures and people who may be end-users and producers/consumers of information used by the system. Socio-technical systems normally operate in a 'systems-rich' environment where different systems are used to support a range of different processes. Figure 1 shows some examples

of socio-technical systems and contrasts these with other types of system which may be complex but which we do not consider to be socio-technical systems.

Socio-technical software intensive systems	Other types of system
Air-traffic control system	Real-time image enhancement system for a radar display
Insurance claims processing system	Spreadsheet
Travel booking system	Flight database
Automated packaging system	Conveyor belt
Company communication system	Telephone exchange switch

Figure 1 Different types of system

The critical difference between the different types of system shown in Figure 1 is that, for socio-technical systems, people, policies and processes of use are part of the system. Socio-technical systems are not just technological artefacts but also involve people who may make mistakes, who will all have different goals, who will make mistakes and behave inconsistently, etc.

Students undertaking computer science and software engineering courses are often techno-centric and cannot see beyond technology to the effects of that technology in use. Our rationale for considering socio-technical systems is that we believe that most systems engineering problems arise because of unexpected interactions between hardware, software and people and we wish to focus our students' attention on these interactions.

In our context, therefore, 'systems engineering' is the activity of understanding, specifying, designing, integrating, testing and deploying socio-technical systems. It is mostly concerned with coarse-grain abstractions and the overall properties which emerge when these abstractions are combined into a system. Hardware and software must be considered but, equally, systems engineering must also be concerned with human, organisational and political problems which are major influences on almost all large systems.

3. Introducing systems engineering in a computer science course

The majority of systems engineering courses which are currently offered are graduate courses which have evolved from 'traditional' engineering courses such as courses in mechanical and electrical engineering. These courses usually include an element of software engineering but their focus is on hardware issues and overall problems with systems hardware such as packaging, vibration and electro-magnetic compatibility.

It is neither appropriate nor realistic to introduce this model of systems engineering into a computer science course. Indeed, I argue that this model of systems engineering where software is peripheral rather than central is outdated given the critical role of software in large, complex systems. There is a need for 'traditional' systems engineering courses to evolve so that they are more concerned with software and human issues.

Our goal is to sensitise students to the problems of systems engineering and to help them expand their perspective so that they can think in terms of systems and not just software. We want to give students enough background that they can get involved in the

systems engineering process and to learn to think in terms of systems as a whole and not just about the software in these systems.

Before going on to discuss how we achieve this, we need to introduce our overall course structure and discuss, more specifically, how this has changed to accommodate systems engineering courses. At Lancaster, departments have a great deal of flexibility in designing course structures ranging from a very flat structure with many options to a hierarchical model where students have very little choice. The computer science course is an example of this latter type of structure where the majority of topics are core topics and must be taken by all students. Our computer science course is structured around three themes with practical project work cutting across and integrating these themes (Figure 1). Students must take 7 courses from each theme plus three further options.

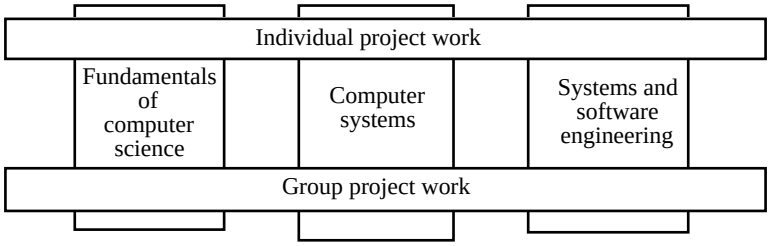

Figure 2 Course structure

In the fundamentals theme we cover topics such as programming in Java, algorithms, databases, formal languages, programming languages and compiler design. In the computer systems theme we cover operating systems, computer systems architecture, communications and distributed systems. High-level hardware design is also covered in this stream although this is not an important component of our course.

Within the systems and software engineering theme, there are four courses which are clearly software engineering courses and two courses which are systems engineering courses. There are three other courses which are primarily concerned with software issues but where we introduce, where possible, broader issues of systems engineering. Figure 3 shows the courses which we offer in this theme:

Software engineering	Software and systems engineering	Systems engineering
Program and data structure design	Interactive systems engineering	Software-intensive systems engineering
Introduction to software engineering	System requirements engineering	Critical systems engineering
Software design	Project management	
Formal software specification		

Figure 3 Systems and software engineering courses

The software engineering courses in the left-hand column are fairly standard courses based around information hiding, the software process, object-oriented design and formal specification in Z. The courses in the middle column cover topics such as user interface models and user interface programming, the requirements engineering process and methods for requirements engineering, configuration management, cost and schedule estimation and process improvement. These courses clearly have a bias towards software engineering but, wherever appropriate, we broaden the discussion to include systems engineering issues.

In the remainder of this paper, we will focus on the courses at the right of Figure 3 namely software-intensive systems engineering and critical systems engineering. These are the courses which are least likely to be familiar to readers. The course on software-intensive systems engineering is a prerequisite for the critical systems course.

3.1 Software-intensive systems engineering

This is the first course that students encounter in the systems and software engineering stream where we don't simply concentrate on software issues. Students taking this course have already taken courses in program and data structure design, introductory software engineering and software design. In introducing the course, we thought carefully about what we were trying to achieve and came to the conclusion that the fundamental goal of the course was what Checkland (Checkland 1981) calls *systems thinking.* That is, we wanted to expand the horizons of students so that they thought of problem-solving in *systems* and not just in *software* terms. When faced with a problem situation, we wanted them to be able to think critically about the problem as well as the possible solutions to the problem.

This was our principal goal for the course but, in addition, we wished to cover specific activities which consume a great deal of resources in the systems engineering process. In particular, we felt it was important to discuss system procurement or acquisition and system integration. These are critically important activities yet there is little or no teaching material available and they are not covered in most software engineering courses.

The current course involves 4 hours of class contact per week for 5 weeks and is structured into two major parts which focus on systems engineering problems and solutions (12 hours) and on systems engineering processes (8 hours).

3.1.1 Systems engineering problems and solutions

In this part of the course, we introduce the idea of complex, software-intensive systems and the process of systems engineering. We discuss where systems engineering and software engineering are comparable and where they are distinct. In line with our objective of encouraging students to develop 'systems thinking' skills, we discuss the analysis of problems and solution strategies and the derivation of high-level system architectures.

The particular topics which are discussed are:

- *Introduction to systems engineering*: Socio-technical systems and the role of systems in organisations. Business goals and business processes. System decomposition - sub-systems and components. An introduction to the systems engineering process and a comparison with the software process.
- *Designing computer-based systems*: Problem analysis and understanding including relationships with strategic goals, stakeholder identification, alternative solution strategies, winner and loser analysis. Feasibility studies including systems comparison, operational constraints, technology assessment, legacy system and process compatibility

and cost-benefit analysis. System specification including hardware/software/process trade-offs, sub-system identification and architectural design specification.

To illustrate these topics, we base the course around a major case study which also serves as a source of practical examples for the students. This case study is a road pricing system which has been proposed as a possible solution to the problems in the UK of growing traffic congestion. In Appendix 1, we include the initial description of the system which is distributed to students. In short, we envisage that users of motorways (in US terms, these are roughly comparable to freeways) pay according to distance travelled with fees collected by a computer-based system. It is a requirement that the system should be based on non-stop operation (that is, no stopping at toll gates) and that it should be installed and operated by a private company rather than central or local government.

This is a particularly good example for this part of the course for a number of reasons:

- All students have had experience of traffic congestion and can understand the domain of operation of the system.
- It is a technically challenging system to develop yet the individual technologies required to develop system components are now available.
- It provides a basis for general problem discussion (is the real problem the development of the system or traffic congestion, what are alternative ways to tackle traffic congestion, etc.)
- The high-level architecture of possible solutions is fairly simple and can be understood by students.
- Performance estimates based on traffic analyses may be illustrated.
- There are significant social and political issues associated with this system such as security and privacy and the impact of the system on local and national economies which can be discussed.

3.1.2 Systems engineering processes

Students are introduced to the systems engineering process in the first part of the course and, in this component, we look in more detail at two of the activities in this overall process. Other activities such as requirements engineering are covered in other courses. We decided to cover systems acquisition and systems integration and deployment because of their critical importance in the systems engineering process and because students did not encounter any comparable topics elsewhere in our course.

The particular topics which we cover in these courses are:

- *Systems acquisition*: Custom designed and off-the-shelf systems (COTS). The different acquisition processes for these different types of system. Product selection. Supplier assessment and management. Legal and contractual issues. Intellectual property.
- *Systems integration and deployment*: The systems integration process. Classes of integration problem (management problems, interfacing problems, configuration problems). System testing, stress testing and acceptance testing. Systems installation planning. Physical, technical and human problems arising during systems installation.

Again, we use a case study to illustrate different aspects of the course and as a source of student problems. The road pricing system is too complex and too far away from the immediate experience of students for this part of the course so we have invented a case study based on a virtual Hellenic museum (see Appendix 1). This is a system involving the creation of a system which links museums in the UK and in Greece and which offers visitors a range of virtual museum experiences. We chose this partly because we have a significant number of Greek students but also because it is a rich source of examples of the real procurement and integration problems which can arise:

- It can be created using off-the-shelf components but there are technical problems in acquiring and integrating components from different manufacturers.
- It involves different organisations from different cultures so problems of interacting procurement processes can be discussed.
- The human problems of introducing such a system into a traditionally conservative environment can be discussed.
- There are problems in testing such a system (how do you test a virtual world?) and in establishing an acceptance test which clearly demonstrates that the system is acceptable.

A general criticism of this course might be that it focuses on problems and not techniques for solving these problems. It does not give students a conceptual toolbox for designing complex systems. One reason for this is that we don't have enough time to cover methodologies such as Soft Systems Methodology (Checkland and Scholes 1990) or Contextual Inquiry (Holtzblatt and Beyer 1993) which are applicable when designing socio-technical systems.

However, even with more time, we would still adopt a problem focus in this course. We think that too many software engineering and computer science courses focus exclusively on 'neat solutions' such as object-oriented design or formal system specification. These often don't take into account the complex and messy reality of the real world. We believe that our concentration here on wicked problems (Rittel and Webber 1973) with no single or simple solution helps students to develop their critical faculties and to get some insight into the limitations of the techniques which we cover elsewhere in our course.

3.2 Critical systems engineering

This course follows the course on software-intensive systems engineering which introduces the problems and difficulties of systems engineering. Here we focus on specific techniques which may be used for the development of systems which are safety, mission or business critical. Again, we take a systems and not just a software focus although there is more emphasis on software in this course.

The course has more time available (30 hours over 10 weeks) than the software-intensive systems engineering course so a more detailed study of problems and techniques which may be applied to solve these problems is possible. We structure the course into 3 themes - real time control systems (8 hours), safety-critical systems (10 hours) and systems reliability (12 hours). The topics covered in each of these themes are:

1. *Real-time control systems* Properties and architecture of real-time systems. Timing analysis and specification. Hardware/software trade-offs. Real-time software design.

2. *Safety-critical systems* Basic principles of safe operation. Techniques for hazard analysis and safety specification. Risk assessment and risk minimisation. Human error classification and designing for minimal error operation. The safety-critical systems development process. Safety verification.
3. *Systems reliability* The relationship between reliability and safety, reliability specification and metrics. Reliability measurement and growth modelling. Fault-tolerant systems architectures. Techniques for software fault avoidance and fault tolerance.

As well as being important topics in their own right for software engineers, we have included these themes in this course because they help students develop their understanding of systems engineering concepts. For example, we can discuss the notions of safety and reliability from a systems perspective and can show that software can be used to cope with failures in other parts of the system and, conversely, how software failures can be tolerated by designing safeguards into the system as a whole.

The material in the course is supported by a major case study which has been drawn from an industrial research project in which we were involved (Sommerville, Sawyer et al. 1998). This is a study of a train protection system for commuter trains on busy suburban lines. This system will stop a train if it passes a red signal or if the speed limit for a track segment is exceeded. This system is ideal as a case study for this course because it is:

- within the experience of students - most of them have travelled on suburban trains,
- a complex socio-technical system with different stakeholders and operational processes,
- clearly a real-time system with potential for software hardware trade-offs,
- a safety-critical system where system failure can result in an accident,
- a system with specific reliability and availability requirements which are separate from the safety requirements. Because of the high density of traffic, it is important that the system does not stop trains unnecessarily.

A current weakness of the course is that it does not include any coverage of security issues which are becoming increasingly important for software-intensive systems. This is partly due to lack of time and partly due to my lack of knowledge in this area. We hope to revise the course in the near future to include some security issues.

3.3 Ethical and professional issues

Along with many other universities, we have recognised the importance of exposing our students to ethical and professional issues as an integral part of their education. Previously, we addressed this in a rather disjointed way with class discussions on ethics and professionalism in various courses but with no links between them. However, when we re-designed the course to give more emphasis to systems engineering, we found that we had created a structure which allowed us to discuss ethical and professional issues in a much more coherent way.

The reason for this is that ethical issues in a software context are unreal. Issues such as the responsibilities of software engineers to ensure safety and privacy cannot simply be confined to software. Software on its own cannot injure people, damage property or reveal personal details to snoopers. It is only when it is embedded in a system that problems arise

and these problems are often related to the social and organisational environment in which the system is developed and used.

Covering these issues in a systems context where political and organisational factors are discussed is much easier. Furthermore, as software acquisition is an inherent part of the course, it is natural to discuss topics such as privacy, data protection and software contracts as part of this course.

Examples of topics which are covered are:

1. Privacy and security are discussed in conjunction with the road pricing system example. This is very appropriate as clearly any system which maintained records about a car's position could be used for tracking the movement of individuals.
2. Contracts, public health and safety and relationships between employers and employees are discussed in conjunction with the virtual museum system. Issues of safety are important as it is a virtual reality system used by the general public, it involves contractual relationships between different partners and is developed by freelance staff who may work for different employers in the same application domain.
3. Safety and professionalism are discussed in conjunction with a discussion of the development of fly-by-wire software for civil aircraft. This is illustrated using a video tape of a television documentary on problems with this type of aircraft.

We do not have separate lectures on ethical and professional issues. Rather, we integrate discussion of these issues with other lectures and, in particular, with class discussions about the practical coursework associated with each of the case studies.

4. Challenges and problems

The challenges which we faced in introducing these changes to our software engineering course fall into two categories:

1. *Pedagogic challenges* - how do we present the material in a way which is relevant to students and potential employers and, at the same time, maintains academic rigour.
2. *Practical challenges* - how do we provide students with appropriate teaching material and practical experience.

The first of these challenges is particularly difficult. Systems engineering is a broad rather than a deep topic and it is difficult to avoid discursive teaching which does not lead to any real understanding. While students might find a selection of 'war stories' entertaining, these would do little to develop their abilities to think in systems terms and would not have the required academic rigour which we expect from advanced courses in a computer science degree scheme.

We address this issue in two ways:

1. The use of case studies, as already discussed, allows us to link different material in the course and illustrate its relevance to reality. Relating the taught material to these case studies and following this with case study related coursework does, we believe, help students develop the ability to think in systems terms - the primary goal of our course.

2. We avoid discussing specific solutions which have been used in specific systems and always argue that the identified problems have several possible solutions. In our experience, students find it very difficult to generalise from a specific solution to a problem and their understanding of the problem is limited to understanding the techniques used in its solution. We deliberately spend most time discussing systems engineering problems. These are timeless and will be similar in 10 or 20 years time. The same cannot be said for many specific techniques which are currently covered in computer science courses.

The most significant practical difficulty which we faced when introducing systems engineering concepts into our courses was the lack of available teaching material. The only book in this area which focuses on software-intensive systems (Thomé 1993) includes useful descriptions of various aspects of systems engineering but is written for the practitioner rather than the student. There are various papers which we make available to students (Andriole and Freeman 1993) (Thayer 1997) but, again, these have not been written for undergraduate students. With a small number of exceptions such as Leveson's book on safety-critical systems (Leveson 1991) most papers concentrate on software and not broader systems issues. In some areas, such as systems acquisition and integration, we could not find any material at all which was not written from a US Government or military perspective.

To address the problem of lack of teaching material, we had no alternative but to create our own notes based on what material there is available and our experience of systems engineering. We supplement these, wherever possible, by recommending various papers as additional reading. Students are issued with paper copies of the notes and also have access to electronic versions in MS Powerpoint format[1].

A problem which we currently face is the difficulty of supporting practical skill development. It is hard enough to identify realistic software projects for students let alone systems engineering projects involving hardware, software and people. Clearly, the scope here is limited but we hoped to give students some laboratory experience in systems integration and in reliability and safety-critical systems development

We have not yet found a good way to do this. One idea which we had to support practical work in systems integration was to take some PCs apart and ask students to put them together again in a particular hardware/software configuration. Our system manager was not enamoured with this idea as he would have to sort out the inevitable problems. Consequently, we decided that, rather than use machines which were shared with other courses, we would use PCs which had been replaced and which were no longer needed in our laboratories. This idea foundered because of the difficulties of obtaining hardware for these machines (the cards were no longer stocked) and because our programming language had changed. Students didn't know the languages for which we had compilers and we couldn't buy compilers for the languages which the students knew.

The difficulties which we face in trying to provide some practical experience in reliable and safety-critical systems development is developing this type of system takes a long time. Rigorous methods are used and, even for small systems, the time taken is more than the single semester which is available for the course. Furthermore, it is difficult to find small,

[1] Copies of PowerPoint slides and associated notes can be downloaded from http://www.comp.lancs.ac.uk/computing/resources/ser/syseng/

self-contained systems in this area which do not require very specialised domain knowledge. We have some ideas of developing a set of simulators for an autonomous vehicle which would provide a basis for practical work but these have not yet come to fruition.

Because of the difficulties of setting meaningful practical work, student coursework is based around paper-based design exercises. The case studies used in our course are initially presented as problem descriptions. Different aspects of these problems are used to illustrate the course material and students are set coursework related to these problems. Some examples of coursework which we set are:

1. Students are asked to design a logical and physical architecture for the road pricing system showing major hardware and software sub-systems and discussing possible physical organisation of the system.
2. Students are asked to write a short essay on the topic of privacy and computer systems and to illustrate this using examples drawn from the road pricing system.
3. Students are asked to complete a system procurement exercise for the museum system and to define equipment requirements and costs for museums in the consortium.
4. Students are asked to assess the performance requirements for the train protection system based on the train speed and the cycle time of the on-board train control system.

Students generally agree that, after they have completed the exercises, they have been valuable learning experiences. They find the coursework to be challenging but also to be disturbing. They don't have any methods which they can use to evaluate the solutions they propose yet they know intuitively that their solutions are inadequate in some respects. They get no sense of closure in that they there is no clear end-point to the work as there is when a program has to be developed.

Educationally, we believe that this is valuable experience. This situation reflects reality where trade-offs and compromises have to be made, resource and schedule constraints define when one phase of the work stops and the next phase begins, there are known problems with systems, etc. However, the instructor must recognise that this is a new experience for students who are used to tractable problems with simple solutions. We have found that students must be given constant reassurance (e.g. by commenting on drafts of their work) throughout the course that they are doing the right thing.

5. Conclusion

This paper has presented an argument that students studying software engineering should be exposed to broader systems issues to help them understand the context in which software operates. We have described how we have designed the software engineering component of a computer science degree to include courses on computer-based systems engineering. These courses are based around case studies and cover topics such as systems design, acquisition and integration, real-time systems, and system safety and reliability.

As well as providing students with a better understanding of real-world problems, we believe that the integration of software and systems engineering in our course has had two further positive benefits:

1. *We can be proud to be software engineers* The systems engineering focus allows us to explain why software problems occur and it means that we can present software engineering as a way to solve systems problems rather than as a problem in its own right. The message that software is a problem has always seemed to us to be demotivating for students. We believe it is educationally much better to present the subject in a positive rather than in a negative way.
2. *Human, social and political factors are made real to students* Introducing systems engineering shows students that there is not necessarily a programming solution to all problems and that non-technical influences are the important influences in real systems engineering projects.

The course changes were introduced in 1996/97 and the first group of students who were exposed to these systems engineering courses have just graduated. With a small number of exceptions, the student feedback to the courses was very positive. The few employers that we have consulted were also positive about our course changes. Although we have a fundamental problem in introducing practical laboratory work, so far we believe that our integration of systems and software engineering has been fairly successful. In short, we believe that introducing systems engineering makes our students better software engineers.

6. References

Andriole, S. J. and P. A. Freeman (1993). "Software systems engineering: the case for a new discipline". *BCS/IEE Software Eng. J.* **8,** 3: 165-78.

Checkland, P. (1981). *Systems Thinking, Systems Practice*. Chichester, John Wiley & Sons.

Checkland, P. and J. Scholes (1990). *Soft Systems Methodology in Action*. Chichester, John Wiley & Sons.

Gibbs, W. W. (1994). "Software's Chronic Crisis". *Scientific American* **271,** 3: 72-81.

Holtzblatt, K. and H. Beyer (1993). "Making Customer-Centred Design Work for Teams". *Comm. ACM* **36,** 10: 93-103.

Leveson, N. G. (1991). "Software Safety in Embedded Control Systems". *Comm. ACM* **34,** 2: 34-46.

Pressman, R. S. (1996). *Software Engineering: A Practitioner's Approach, 4th edition*. New York, McGraw-Hill.

Rittel, H. and M. Webber (1973). "Dilemmas in a General Theory of Planning". *Policy Sciences* **4,** : 155-69.

Sommerville, I., P. Sawyer, et al. (1998). "Viewpoints for requirements elicitation: a practical approach". *Proc. Int. Conf. on Requirements Engineering.*, Colorado.

Swartz, A. J. (1996). "Airport 95: Automated Baggage System?". *ACM Software Engineering Notes* **21,** 2: 79-83.

Thayer, R. H. (1997). “Software System Engineering: An Engineering Process”. *Software Requirements Engineering*. R. H. Thayer and M. Dorfmann (eds). Los Alamitos, Ca., IEEE Press.

Thomé, B., Ed. (1993). *Systems Engineering: Principles and Practice of Computer-based Systems Engineering*. Chichester, UK., J. Wiley and Sons.

7. Appendix 1

This appendix includes the problems descriptions which are handed out to students during the systems engineering course and which are used as the basis for the case studies. These are deliberately written in a high-level way and we know that we are incomplete. This lack of detail means that different student groups have the flexibility to tackle the coursework in different ways.

7.1.1 Problem description - A road pricing system

A general problem which we face in the UK is that the current rate of growth of private car ownership is unsustainable. We must find a way of encouraging people to make more use of public rather than private transport.

We have a fairly extensive motorway network and a very high proportion of personal and goods transport uses this network. Significant investment is needed to maintain the motorway network in good condition.

It is government policy to slow down and (ultimately) reverse the growth in private car ownership. It has been proposed that the motorway network should not be significantly extended and that a policy of 'road pricing' should be introduced where users of the motorway network pay according to the distance travelled. It is also Government policy that such an initiative should be based on private finance and the system should be procured and managed by private companies rather than government organisations.

Of course, road pricing in some form is not a new idea. The French motorway system is a toll-based system and tolls are also payable on some US roads. However, introducing a system based on toll gates where payment is made is not acceptable in this case for two reasons:

1. The traffic density and patterns on UK motorways is so high that any system which required drivers to stop and pay tolls would inevitably create unacceptable traffic congestion and delays.
2. Toll-based systems have relatively high running costs because completely automatic operation of systems where payments must be made directly is not possible.

An alternative to a toll-gate based system is a completely automated road pricing system which detects the presence of vehicles and automatically levies a charge when they enter a motorway segment. This is clearly a complex computer-based system which involves roadside vehicle detection systems, charging databases and mechanisms for drivers to pay for their road usage.

The following assumptions may be made about the design of this system.

1. Individual vehicle transmitters allowing vehicles to be individually identified at current traffic volumes are available at reasonable cost.
2. While you can expect more than 95% of vehicles using the motorway to have such transmitters, a significant number of vehicles won't be equipped and some mechanism must be devised for charging these vehicles for road use.
3. Vehicle number plates can be recognised using current techniques of image analysis.

4. It is not an absolute requirement that the system should be continually available. Loss of service will mean some loss in revenue but so long as revenue is at least 95% of the possible maximum this is acceptable.
5. Variable pricing may be used to try to even out the load on the motorway network. For example, travelling at peak hours may be more expensive than travelling at other times of the day.

7.1.2 Problem description - A virtual Hellenic museum

A consortium of museums wish to collaborate to produce a 'Virtual Museum of Ancient Greece'. This is a multimedia system which provides historical information about the culture and artefacts of Ancient Greece, information about existing Greek antiquities, information about archaeological digs to recover information, etc. The consortium includes museums in both the UK and in Greece which have important collections of Greek antiquities.

The key aims of the project are:

1. To produce a multimedia database which includes information about the major items in the collections of the participating museums. This is intended for use by casual users in any of the museums.
2. To allow scholars in any of the museums to have access to the complete catalogues in all other museums and to provide a means of communicating with museum curators and historians in the museums.
3. To provide a number of virtual reality simulations of life in Ancient Greece so that museum users can gain an impression of life in these times.

The curators of the museums are attracted by this idea but do not have any detailed knowledge of the technology to be used. Different museums have different budgets for equipment and software ranging from £12, 000 to £250, 000. The museums with the lowest budgets require at least 2 terminals which provide access to the system and do not require (but would obviously like) virtual reality simulations. Other museums require either 6 or 12 terminals with access to the system. Some of these might be used for virtual reality simulations but, if appropriate, additional virtual reality systems may be provided.

It is anticipated that each museum will prepare data for the project from its own collections. In addition, one person will be employed for 1-2 years to procure and set up the system.

7.1.3 Problem description - A train protection system

To increase the safety of its system, a train operating company has decided that it wishes to install a train protection system in its commuter trains which carry passengers from the suburbs to the centre of a major city. A train protection system is a safety system which protects against driver error. It will bring the train to a halt under the following circumstances:

1. When the train exceeds the speed limit for the current segment of track.

2. When the train fails to stop at a danger signal or passes a caution signal at too high a speed.

The train protection system is closely integrated with the railway signalling system. The signalling system is based on the notion that the railway line is split into a number of track segments with signalling equipment controlling the entrance to each track segment. Safety in the system is assured if it is impossible for there to be more than one train in a track segment at one time and maintaining this constraint is the fundamental objective of the signalling system.

However, the designers of the signalling system must also take into account the fact that trains cannot stop instantaneously and that collisions could conceivably occur when trains are signalled in different segments but the braking distance of a train is such that it could run into the train in front. Therefore, track segments may have additional signals and may have a specified speed limit which depends on the length of the track segment and the specific segment characteristics such as the gradient, curvature, etc.

The train protection system consists of a number of sub-systems:

1. On-board hardware and software which is installed on the train and which is integrated with the existing train control system.
2. Trackside equipment which is controlled by the signalling system and which sends the signal status and the track segment speed limit to the on-board system.
3. A signalling centre system which receives signalling information from the existing signalling system and communicates with the trackside train protection equipment.

A constraint on the development of this system is that there already exist systems for signalling and train control and the train protection system must be installed without disruption to these systems.

Figures

Socio-technical software intensive systems	Other types of system
Air-traffic control system	Real-time image enhancement system for a radar display
Insurance claims processing system	Spreadsheet
Travel booking system	Flight database
Automated packaging system	Conveyor belt
Company communication system	Telephone exchange switch

Figure 4 Different types of system

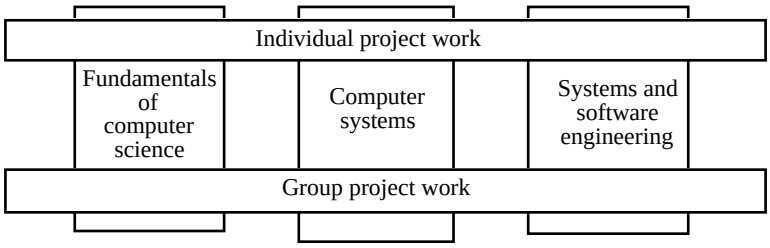

Figure 5 Course structure

Software engineering	Software and systems engineering	Systems engineering
Program and data structure design	Interactive systems engineering	Software-intensive systems engineering
Introduction to software engineering	System requirements engineering	Critical systems engineering
Software design	Project management	
Formal software specification		

Figure 6 Systems and software engineering courses