
# Committee for the Evaluation of Software Engineering and Information Systems Engineering Study Programmes

## The Technion

## Faculty of Computer Science
## Software Engineering Track

## Evaluation Report

## Contents



## Appendices

## Chapter 1: Background

The Council for Higher Education (CHE) decided to evaluate the study programmes in Software Engineering and Information Systems Engineering during the 2013 academic year.

Following the decision of the CHE, the Minister of Education, who serves ex officio as Chairperson of the CHE, appointed a review committee consisting of:

- Prof. David Parnas (Emeritus) – Engineering, McMaster University, Canada - Committee chair
- Prof. Carl Landwehr - Cyber Security Policy and Research Institute, George Washington University, USA
- Prof. Jochen Ludewig - Chair of Software Engineering, Stuttgart University, Germany
- Prof. Robert Meersman, Department of Computer Science, The Vrije University - Brussels, Belgium
- Prof. Peretz Shoval – Department of Information Systems Engineering, Ben Gurion University, Israel
- Prof. Yair Wand - Sauder School of Business, The University of British Columbia, Canada
- Prof. David Weiss1 - Department of Computer Science, Iowa University, USA
- Prof. Elaine Weyuker- Visiting Scholar, DIMACS, Rutgers University, USA

Ms. Daniella Sandler served as coordinator of the committee on behalf of the CHE.

Within the framework of its activity, the review committee[2] was requested to:

1. Examine the self-evaluation reports, submitted by the institutions that provide study programmes in Software Engineering and Information Systems Engineering and to conduct on-site visits at those institutions;
2. Submit to the CHE an individual report on each of the evaluated academic units and study programmes, including the committee's findings and recommendations;
3. Submit to the CHE a general report regarding the examined field of study within the Israeli system of higher education, including recommendations for standards in the evaluated field of study.

The process was conducted in accordance with the CHE's October 2012 Guidelines for Self-Evaluation.

[1] Professor Weiss was not present for the visits to the Technion.

[2] The review committee's letter of appointment is attached as **Appendix 1**.

## Chapter 2: Committee Procedures

The review committee held its first meetings on April 26, 2013, during which it discussed fundamental issues concerning higher education in Israel, the quality assessment activity, and Software Engineering and Information Systems Engineering study programmes in Israel.

From April 28th to May 7th, 2013, the review committee visited Ben Gurion University, The Technion and ORT Braude College. From June 6th to June 14th 2013, the review committee visited Afeka College, Shamoon College of Engineering (SCE), Shenkar College, Jerusalem college of Engineering and the Jerusalem College of Technology. During these visits, the review committee met with various stakeholders at the institutions, including management, faculty, staff, and students.

This report deals with the Software Engineering Track in the Faculty of Computer Science at the Technion. The committee's visit took place on May 1, 2013.

The schedule of the visit is attached as Appendix 2.

The review committee thanks the management of The Technion and the teachers of the Software Engineering Track in the Faculty of Computer Science for their hospitality towards the committee during its visit at the institution.

## Chapter 3: Executive Summary

This report evaluates the Software Engineering Track at the Technion and makes recommendations for improving it.

Recognizing that there is no widely accepted standard that specifies what should be taught in Information Systems Engineering and Software Engineering programmes, Chapter 4 of this report explains our committee's consensus on the subject; it explains "Software Systems Engineering" as *the multi-person development of multi-version programs* and then details the implications of this description for the content and organization of Software Engineering programmes and Information Systems Engineering. A principal conclusion of the report is that Software Systems Engineering includes more than programming. The remaining chapters of the report use the criteria explained in Chapter 4 to evaluate the Software Engineering Track at the Technion.

At the Technion, the Software Engineering programme is a track taught by the Faculty of Computer Science; it is run by a six person committee.

Our committee found that the present programme is best described as a Computer Science programme that stresses programming rather than a Software Engineering programme as described in Chapter 4 of this report. In other words, the programme focusses on programming rather than the full process of building software.

The following points provide a brief summary of our evaluation.

- We found the mission statement to be too generic and not specific to the problems of Software Engineering.
- We noted that there was no course that offers an overall introduction to Software Engineering.
- The SE methods course is badly overloaded. There is insufficient time to learn to use the methods.
- There is no course on Human-Computer Interfaces (or Human-Computer Interaction).
- Computer Security does not get enough attention.
- The capstone project is a weak point of the programme.
- Software Quality Assurance (SQA) is not adequately covered by the programme.
- Courses on data bases, computer system security, project management, and software design should be required of all SE track graduates but are not in the present programme.
- The committee was positively impressed by the quality, energy, and enthusiasm of the Technion's CS faculty. The faculty members are doing excellent and interesting research but they are not doing research in areas that the committee considered to be core SE areas.
- The faculty that we met take their teaching obligations seriously and are accessible and supportive of students.
- The students that we met were excellent students, well prepared, smart, vocal, focussed and enthusiastic. Graduates have been well received by industry.

- We noted a lack of formal studies about the extent to which the stated learning goals are being achieved.
- Building, classrooms, library and other such facilities were adequate.
- The committee found that the self-evaluation report was hard to use for our purposes. Too much of it was devoted to bragging (often in the form of pictures). It was difficult to find the information that was relevant to our task. We wanted to see more of the information that was given to students as well as more information on course content.

## Chapter 4: Evaluation Criteria for System Software Engineering Programmes

### 1. Purpose

To fully understand this committee's observations and suggestions, it is necessary to know how we understand the terms "*Software Engineering"* and "*Information Systems Engineering"*. Although these terms are widely used, there is no widely accepted definition of either. This report reviews discussions that took place when the terms were first introduced[3] and uses them to explain how we view the goals of *Software Engineering* and *Information Systems Engineering* programmes.

It is not our purpose to describe or prescribe a particular curriculum; instead, this chapter explains some of the capabilities that one needs to work as a fully qualified professional in software intensive engineering. Many, quite different, curricula could achieve the learning outcomes that we describe.

The fields of *Software Engineering* and *Information Systems Engineering, which* developed separately, are two members of a general class of disciplines that we call "*Software Systems Engineering*" (SSE). We use this term to encompass all engineering disciplines in which software is the dominant technology.

- Section 2 of this chapter relates discussions that took place when the term "Software Engineering" was first introduced.
- Section 3 discusses some capabilities that Software Systems Engineers need.
- Section 4 discusses the role of projects in Software Systems Engineering education.
- Section 5 illustrates a few of the many distinct disciplines that are included in Software Systems Engineering.
- Section 6 provides a more detailed discussion of important learning outcomes for information systems engineering.
- Section 7 discusses how the previous sections can be used when designing or revising a curriculum.

### 2. Searching for a definition of "Software Systems Engineering"

In the early 1960s, some computer scientists began to use the phrase "Software Engineering"[4], without providing a clear definition [2, 3]. In doing so, they expressed the hope that software developers would learn to construct products with the discipline and professionalism that they associated with professional engineers.

When the term "Software Engineering" was first introduced, many asked, "How is that different from programming?" More recently, when post-secondary "Software Engineering" programmes were introduced, some asked, "How is that different from Computer

[3] late in the 1960s through the early1970s

[4] Historically, the term "Software Engineering" was used. However, we believe that what is said in this section applies to all Software Systems Engineering disciplines.

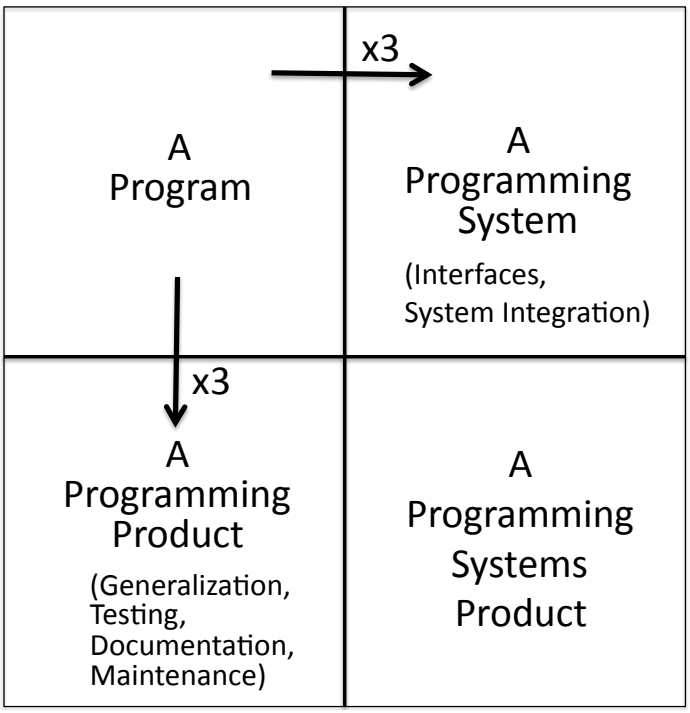
Science?" Some were questioning the need for a new term; others wanted to know what, beyond programming and computer science, would be taught to students of software engineering. Two simple, but consequential, answers emerged. Although both definitions are old, they have withstood the test of time; they are consistent with current usage, and remain relevant today.

## 2.1. Brian Randell's answer

One of the best answers to the question was provided by Prof. Brian Randell, one of the organizers of the first two Software Engineering conferences [2,3] and co-author of two frequently referenced reports on those meetings. In discussions, he has described software engineering as "multi-*person* development of multi-*version* programs". This pithy phrase implies everything that differentiates the practice of software engineering from advanced programming. Software engineers must be able to work in teams to produce programs that will be used, and often revised, by people other than the original developers. Performing that job professionally requires a mastery of programming, but many other capabilities are required as well.

## 2.2. Fred Brooks' answer

The diagram below appears In Fred Brooks' classic book, "The Mythical Man-Month" [1]. It has two dimensions; we call the vertical dimension "productizing" and the horizontal one "integration".




***Fred Brooks' explanation of why software engineering is more than programming.[5]***

[5] Figure redrawn from [1]. The "x3" annotation, denotes a 3-fold increase in effort.

By integrating a program with other, separately written, programs, one moves from a program to what Brooks called “a programming system”. By testing, documenting, and preparing a program for use and maintenance by other people, one converts that program to “a programming product”. Doing both results in a “programming systems product”. Going from a program to a programming systems product results in a massive increase in cost and effort.

Brooks’ formulation, like Randell’s, makes it clear that there is much more required of a software engineer than programming skill. Software engineers must master programming, but they must also be able to integrate separately written programs and “productize” the result.

## 3. What should Software Systems Engineers be prepared to do?

The decision by Israel’s Council for Higher Education to evaluate “Software Systems Engineering” programmes separately from “Computer Science” programmes makes these old questions relevant today. We have to ask how Software Systems Engineering programmes differ from Computer Science Programmes and what criteria should be applied when evaluating them.

This section lists some activities that are implied by both Randell and Brooks. We believe that Software Systems Engineering programmes should teach the fundamental principles and procedures that will help their graduates to perform these tasks well. We do not prescribe the content of a Software Systems Engineering programme. Instead, we describe capabilities that should be kept in mind by those who design and teach Software Systems Engineering programmes. Moreover, we do not describe everything that should be in a programme; we focus on a core set of capabilities that should characterize Software Systems Engineering programmes.

This report deliberately avoids naming any popular current technologies and methodologies. For example, although we suggest that graduates should have learned how to create and use models, we do not mention any modelling languages or modelling tools. The choice of how, and when, to teach modelling is left to the individual institution.

We have five reasons for not naming specific tools or languages in this report:

- Software technologies change rapidly and quickly become outdated. A post-secondary education should prepare students to learn new technologies as they are developed; graduates should also be prepared to learn an old technology if asked to work on a product that was originally developed using tools that are no longer in common use.
- Many current technologies are commercial products whose usage and effectiveness are exaggerated by experts who act more like sales representatives than like scientists.
- Some instructors will have favourite tools or languages that fit their view of how things should be done; they will want to use those tools in their teaching.
- The development of a new method or tool should not obsolete the education of an engineer who was educated before it appeared.

- It is more important that graduates of a Software Systems Engineering programme are able to use the available evidence to evaluate new methods and technologies and to choose the approach that is best for their present project, than that they have been taught about any particular tool or notation.

However, one cannot teach engineering methods and concepts without giving students a chance to apply them while they can benefit from guidance by their instructors; during their education, students will learn to use some of the current tools. To help students to distinguish between fundamental principles and current technology, it is often useful to separate the teaching into lectures and laboratory sessions so that

- the lectures teach fundamental concepts and principles, and
- the laboratory sessions provide experience applying the lecture material using reasonably current tools and notations.

The committee is quite aware of current developments in Software Systems Engineering programmes and the explosion of topics and terminology that has resulted from the increased level of activity in our field. It would be impossible for us to discuss all of the issues and concepts that have been discussed in the academic and industrial literature. We have chosen instead to focus on issues and capabilities that we consider to be fundamental and hence a core body of knowledge. Each institution should build on this core in its own way.

We expect the notations and tools that are popular today to be replaced by newer tools and approaches. In contrast, we believe that the capabilities discussed below are fundamental and will remain important throughout a graduate's career.

## 3.1. Communicate precisely between developers and stakeholders

For a product development to be successful, the needs and preferences of the stakeholders (e.g., purchasers, clients, investors, present and future users) must be communicated to the developers. It is best if these requirements are determined and documented early in the development process.

Understanding user needs before a system is available is very difficult. Often the clients do not know what is required; even when they know, they may not be able to communicate those requirements clearly. Consequently, their statement of requirements will change during the project and will continue to change throughout the period in which the system is used. Developers must be prepared to respond to the changes that (inevitably) occur both during system development and after deployment. Nonetheless, there are many advantages to having the best possible description of the client's requirements before code writing begins and updating that description to keep it consistent with the client's needs.

As in other engineering fields, determining and documenting requirements requires extensive interaction between potential users (or their representatives) and representatives of the developers; it may require modelling (see section 3.8 of this chapter) and building prototypes (simulators or mockups) that the clients can use and critique. The goal of these

efforts by developers and stakeholders should be to produce a requirements documentation that is precise and demonstrably formally complete[6].

Usually, the stakeholders' representatives will not have been trained in requirements specification. In those cases, the Software Systems Engineers will have to take the lead by producing a draft statement of requirements that can be reviewed, corrected, and (eventually) accepted by the purchasers or user representatives.

After the development is complete, the user-visible properties of the product must be communicated to the end-users so that they can make effective use of the system. This, too, is part of the responsibility of a Software Systems Engineer.

In short, Software Systems Engineers should learn how to elicit and document requirements, how to describe the visible behaviour of a completed product, and should understand the importance of keeping these documents up to date as user requests and system behaviour evolve.

## 3.2. Communicate precisely among developers

When many developers cooperate to produce a software product, the responsibilities of each developer should be clearly specified and understood by other team members; otherwise, the project will be plagued by gaps, duplication, and incompatible components.

Generally, the programming is organized as a set of tasks; each task is to design and develop a module; each module is the responsibility of a group of developers or an individual developer. All developers must know both how their module is required to behave and what can be expected of other modules.

The assumptions that developers of module A make about the behaviour of module B constitutes B's interface to A. Because even small deviations can cause major changes in software behaviour, interface descriptions should be unambiguous and cover all possible usage sequences (i.e., be formally complete[4]).

In short, Software Systems Engineers should learn how to read and write module interface documentation and how to design module interfaces.

## 3.3. Design human-computer interfaces

Software Systems Engineers will be designing systems that present information to, and solicit information from, people. When designing human-computer interfaces they should learn to consider both the nature of the information that will be exchanged and the capabilities and habits of the intended users.

Software Systems Engineers should also learn to distinguish between ease-of-learning and ease-of-use and how to design interfaces for both beginning and experienced users.

---

[6] By "formally complete" we mean only that the documentation characterizes acceptable behaviour in all possible situations. Formal completeness should not be confused with correctness; it does not guarantee that the behaviour identified as acceptable by the document is actually suitable for the intended use. It is possible for a Software Systems Engineer to check a document for formal completeness; one needs stakeholders to check for correctness.

Designing human-computer interfaces is a multidisciplinary subject. It combines knowledge from fields such as psychology, engineering biomechanics, industrial design, and graphic design, with an understanding of software design and the characteristics of interface devices. There are software packages designed to ease the construction of user interface software; a Software Systems Engineer should understand what such products can do and know how to choose the one that is best for their project.

In short, Software Systems Engineers should know how to design human-computer interfaces that are easy to use and that improve the user's productivity.

## 3.4. Design and maintain multi-version software

Successful software products evolve but the older versions don't necessarily disappear; often, the older versions must remain in the development organization's product line. Knowing how to design and maintain software product lines is essential for Software Systems Engineers.

Designing products for ease of change requires approaches that often seem counterintuitive to programmers. These approaches sometimes lengthen the development time for the first version in order to reduce both the time required to develop later versions and the cost of maintaining several versions simultaneously.

There are several approaches to software design that make a product easier to change. All of them require the developers to take the time to think about possible changes during the initial design. Some of the approaches to making software easier to change that can be taught to Software Systems Engineering students are discussed in more detail in the remainder of this section.

### 3.4.1. Separate (changeable) concerns

Software revisions are easier if the software is designed so that the most likely changes are localized in modules that can be revised without affecting other modules. Software Systems Engineering students should learn how to:

- study the requirements to identify the aspects that are most likely to change,
- study the support system and hardware to identify the aspects that are most likely to change,
- study the software design decisions (algorithms, internal interfaces and data structures) to identify the decisions that are most likely to require change,
- organize projects as a set of modules that can be developed, and changed, independently because each module is solely responsible for a changeable aspect of the system, and
- design module interfaces that abstract from the changeable aspects of the module so that the interface is not likely to change when the module implementation is changed.

Experience has shown that even when the changes required were not anticipated, software designed in this way is easier to maintain than software designed without concern for future changes.

In short, Software Systems Engineers should learn to prepare for change by thinking about what is likely to change and encapsulating the most changeable aspects of their work in well-identified modules.

### 3.4.2. Document to ease revision

When change is needed, the code may be revised by people who did not write it. Even if the original programmers do the revisions, they may no longer remember the details and the reasons for the decisions that they made. Consequently, they should leave a detailed and precise design description for the maintainers. Software Systems Engineering students must learn how to:

- produce detailed design documentation that will serve as a “knowledge base” for future developers,
- organize the documentation so that information is easy to find,
- organize the documentation so that the information that is most likely to change is not repeated, and
- maintain all artifacts (both code and documents) in an accurate and up-to-date state so that the documents can serve as a reliable source of information for future developers.

In short, Software Systems Engineers should learn how to produce and maintain orderly design records to help future maintainers.

### 3.4.3. Use parameterization

Software Systems Engineers should learn to parameterize documents and code and avoid including constants that are likely to change in their code.

### 3.4.4. Design software that is easy to “port” to other platforms.

With the wide variety of platforms available today, it is important that software developers be able to develop and maintain products that can be used on several of them. Porting software to new platforms can lead to conflicting requirements and difficult design decisions for the developer. For example:

- Users of a platform may prefer an application to have the same “look and feel” as other applications on their platform.
- Users of a product may want it to have the same “look and feel” on all of their platforms.
- One platform may provide services that make it easy to offer a capability that would be difficult to offer on some other platforms.
- The mechanisms available for transferring data from one application to another may differ between platforms.
- Error handling conventions can differ between platforms.

Industry has had mixed success in solving these problems. Often product lines have features that are not available on some of the platforms. This can confuse and annoy users.

In short, Software Systems Engineers should be aware of the problems of building multi-platform products and understand how to organize software so that the platform-specific components are isolated and easily replaced when a product is ported.

### 3.4.5. Design software that is easily extended or contracted

Users have come to expect software products that are easily extended when new capabilities are needed. They would like the extended product to be compatible with the previous version so that:

- They can continue to use the old capabilities without changing their habits or other programs.
- The extended program can work with data created by the previous version.

In other situations, there may be a need for a reduced capability product. For example, a software company may want to offer a free version to attract interest or a version that will run with limited computational resources.

If some products are extensions of others, the cost of maintaining a software product line will be reduced whenever a change can be effected by revising a shared component.

Extending and contracting software has proven unexpectedly hard for industry to achieve; often, the small platform versions are neither true subsets of the larger ones nor fully compatible.

In short, Software Systems Engineers should learn how to design software so that it can more easily be extended or contracted and data can be transferred between versions.

### 3.4.6. Manage products that exist in many versions

Software development organizations must often maintain many versions of their software products. Those products comprise a large set of components and several variations of each of those components may exist. It can be very difficult to make sure that a particular version of the product contains the right version of each of its components. Making sure that all products have the correct versions of their components is often called configuration management.

There are many configuration management tools available to organize the versions of components and assemble working systems.

In short, students should be taught the basic principles of software configuration management and provided with an opportunity to apply those principles using at least one of the tools.

## 3.5. Design software for reuse

Software reuse is like good parenting; everyone is in favour of it, but it is often very difficult. Unless the software was designed, and documented, with future reuse in mind, it may be harder to reuse a component than to produce a new one. If software is being reused, it must be thoroughly tested in its new environment.

Even when an organization has code that could be reused in a new project, the developers of that product may not know of its existence or capabilities.

Program code is not the only software artifact that developers would like to reuse. Development organizations would also like to be able to reuse documents, and test suites. This too is more easily said than done. Small changes between versions may subtly invalidate previously issued documentation and tests. Older documents may ignore new features or assume the presence of a feature that has been replaced. If a characteristic of the older versions has been mentioned in several parts of the documentation, those revising the document may produce an inconsistent document by changing some occurrences but not others. Test suites can be inadequate for newly added features or can falsely indicate a failure because of a deliberate change. It takes careful structuring of documentation and test suites to make reuse easier.

In short, Software Systems Engineers should learn how to design and document software components to make the code and other artifacts easier to reuse; they should also learn how to organize repositories for the potentially reusable artifacts so that any reusable assets can more easily be found when needed.

## 3.6. Software quality assurance

The fact that software is written for use by people who did not develop it and do not understand its structure, adds to the responsibility of those who develop the software. When we write a program for our own use, we can detect, understand, and respond to failures that occur during use; other users will not have that capability. For them, reliability is especially important.

Software systems engineers should be proficient in methods that help to assure that the quality of a product is acceptable before it is released to customers. In particular, they should be familiar with methods of:

- testing programs — both to find and eliminate faults and to estimate reliability to assist in deciding whether a new product is ready for release,
- inspection of large programming systems,
- formal verification of critical small programs, and
- reviewing documentation for formal completeness and accuracy.

Software Systems Engineers must understand that quality assurance is a professional obligation and that they should refuse to release products that have not been shown to be fit for their intended use. Software systems engineers should also understand how to design software to make inspection and testing easier.

In short, Software Systems Engineers should be taught that they are responsible for the quality of the products that they release to users and must be familiar with basic methods of quality assurance.

## 3.7. Develop secure software

Today's software is used over networks that can be used to attack user's systems. Even software that is not connected to a network is often used by many users; some of those users may try to interfere with others. Even single-user software can, if it has not been

carefully designed, be tricked into behaving in unacceptable ways and inflict damage on its users.

A Software Systems Engineer should know how to design software that does not allow one user to either interfere with, or gain access to the data of, others. They should also know how to make sure that software is robust and can handle maliciously crafted (or erroneous) inputs properly.

Software Systems Engineers must understand that security must be a design concern from the start; many years of research and experience have made it clear that security is not achievable if it is an afterthought.

In short, Software Systems Engineers should understand the basic principles of designing programming systems that are intrinsically secure.

### 3.8. Create and use models in system development

The creation and use of models (physical, mathematical, or diagrammatic) plays an essential role in all engineering disciplines. Because models are simpler than the product, there will inevitably be differences between the product and the model. Consequently, great care must go into creating the model; even greater care is required when interpreting the results of an analysis of a model.

In short, modelling is quite different from programming; Software Systems Engineers should be able to create and analyze a variety of models and use the results to improve the design of the software systems that they develop.

### 3.9. Specify, predict, analyze and evaluate performance

Because today's software systems integrate the work of many people and are often being used by many people at the same time, resource utilization can be hard to predict and the response time of the system is often unsatisfactory. Even systems that compute correct values every time that they are used, will be considered unsatisfactory if they are too slow or require too much memory.

Software Systems Engineers are responsible for insuring that their products perform adequately. They must be able to specify the required performance of a system. Specification of performance requirements requires a characterization of the expected load on the system as well as anticipating the expectations of users. Prediction of the performance of a proposed design may require making models of both the system and its environment and the application of queueing theory and simulation techniques.

If a system is found to have inadequate performance, the Software Systems Engineer must be able to analyze the hardware and software to identify the causes of the problems. When a system is deemed complete by its manufacturer, Software Systems Engineers who work for the customers will need to carry out performance tests before accepting it.

In short, producing systems that perform adequately, and evaluating the performance of such systems, requires Software Systems Engineers to be competent in a variety of techniques, such as queueing theory, that were developed outside of Computer Science.

## 3.10. Be disciplined in development and maintenance

"The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that characterize the established branches of engineering" [3].

If we observe the way that engineers work in fields such as Civil Engineering, we see that they are required to follow rigid processes when designing, documenting, and checking their products. These procedures are often embodied in forms that must be completed, standards that must be satisfied, measurements that must be made, and mathematical analyses that must be performed. Discipline is also required when repairing an engineering product. Engineers are not born with disciplined work habits; they have to be taught.

Analogous procedures need to be learned by Software Systems Engineers; students should be given numerous opportunities (in the form of projects and exercises) to practice the procedures that they have been taught so that they can gain a deeper understanding of the principles behind procedures and develop good habits.

Software Systems Engineers also need to learn how to modify other people's programs in ways that keep them consistent with the original structure of the software. Software maintenance requires at least as much discipline as the original development.

Numerous tools intended to help the development team to work in a careful and disciplined way are on the market. A Software Systems Engineer must understand what these tools can, and cannot, do and know how to choose and use the best tool for a given project.

To improve the discipline of their software development teams, many companies specify a development process, i.e., a sequence of steps that their development teams must follow and a set of work products that must be produced.

Some companies choose a process that has been developed and promulgated by external experts. Those experts may be either academics or commercial consulting firms; many of their processes are publicized and supported by books and articles. Other companies invest time and effort to develop their own process and produce internal documents that describe it. They do this because they believe that their company has characteristics that make the externally developed processes a poor fit for them.

Some surveys and other inquiries have revealed that even when developers say that they are following a specific process, they actually deviate from it by skipping steps, modifying some products, or adding extra work products. Many different processes have been proposed because none is appropriate for every situation. Software Systems Engineers need to be familiar with several processes, understand their intended purpose and the strengths and weaknesses of each.

In short, Software Systems Engineers should be taught to work with care and discipline both when they are developing a new product and when they are modifying an old one.

## 3.11. Use metrics in system development

Because of the complexity of engineering products, they are often evaluated, explained, and sold to others on the basis of “figures of merit” or “metrics”. Many metrics have been proposed for software; experience has shown that many metrics are misleading in that a “better” value does not always mean a better product. Software Systems Engineers should learn what metrics are useful and the main considerations in choosing metrics.

In short, Software Systems Engineers should learn about software metrics and know how to use them with caution.

## 3.12. Manage complex projects

Building and installing software requires coordinating the work of developers, marketers, clients, and users. Coordinating the work of many people to produce a new system is a complex task. It requires:

- planning and scheduling,
- estimation of cost, time and effort needed for a task,
- progress measurement,
- problem tracking,
- risk management,
- configuration management,
- resource control,
- task assignment,
- forming teams and other organizations (e.g. matrix organizations),
- choosing and using project management tools
- leadership

There are two contrasting views about the relation between project management and Software Systems Engineering:

- Project management is an important problem for all engineering disciplines. Project management is often taught by both Management and Engineering faculties; much of what is taught is independent of the nature of the product. Consequently, one can view project management as a discipline that is outside of Software Systems Engineering. Some institutions include a semester course on project management in their general engineering programme. Others offer project management as an extension to an engineering degree (extra year) or offer engineering project management at the Masters level for students who already have an engineering degree.
- Experience suggests that project management is more difficult in software projects than it is in other engineering projects of a similar size. Problems arise in software projects that are not common in other disciplines. Consequently, some believe that software project management should be an integral component of a Software Systems Engineering programme.

Several reasons have been advanced for treating project management differently from the way that it is treated in other engineering disciplines.

- Software project management is made more difficult by the fact that plans and decisions often have to be made using incomplete and insufficient information.
  - The lack of complete information is most noticeable for the important decisions that are made early in the development process when requirements are not well understood and are poorly documented.
  - Requirements that were stated early in the project are often revised later.
  - During the development, internal interfaces are often incompletely documented; as a result, much time is spent coordinating groups and adjusting the interface between separately developed components to make those components compatible.
- Many software products are new and innovative; consequently, estimation of cost and effort is more difficult.
- Often, managers and developers lack experience with a new approach and cannot accurately predict how well their approach will work.
- The structure (aka architecture) of the software is often designed in such a way that changes in one component may seriously affect the completion time of others.
- For software products, there is often no clear end to the development period; development commonly continues after the release of the first version.
- Because a software development organization must often maintain many versions of its software products, those products are composed of a large set of components; further, several variations of each of those components may exist. It can be very difficult to make sure that a particular version of the product contains the right version of each component.

In short, whether one considers project management as an integral constituent of Software Systems Engineering, or views it as a separate topic, every Software Systems Engineer should understand the basics of project management. There is a great deal of substantive material that can be taught in an academic programme.

## 4. The role of projects in Software Systems Engineering education

Professional educational programmes (e.g., engineering, medicine, and law), are designed to teach “how to” as well as “about”. Engineering programmes teach a great deal of science and mathematics but have the additional obligation to make sure that their students know how to apply what they have been taught when solving practical problems.

Because the application of scientific knowledge is so important in engineering, laboratory exercises and small projects should be used throughout the curriculum so that students can experience how the theory that they hear about in lectures can be applied in practice. The importance of these projects is often underestimated. Projects should be treated as a major component of a Software Systems Engineering programme; all work

should be carefully evaluated, and detailed feedback should be provided to students. Instructors must be given sufficient time and resources to do this.

Many engineering programmes use a final-year project (sometimes called a "capstone project") to provide students with an opportunity to consolidate what they have been taught by using that knowledge while they can still get guidance and supervision from their teachers. Such projects allow them to see how the many facts, concepts and procedures that they have learned in earlier years can be combined to form a coherent process for producing products. In a well designed capstone project, all students will play a few roles[7], which will give them a broad view of the product production process.

Because "multi-person" is one of the defining characteristics of Software Systems Engineering, it is important that, in the projects in a Software Systems Engineering program, the student experiences:

- organizing a multi-person project as a set of fully specified individual tasks,
- completing one or more of those individual tasks, and
- integrating separately developed components to produce a high quality product that can be used by others, and
- experiencing the reaction of others to using their product.

In the capstone project, every team member should have several clearly defined tasks, and participate in discussions about project management and interfaces.

If the project is properly supervised, the students will have a valuable learning experience, their completed individual projects can be used by the faculty to evaluate their ability (i.e., give them a grade), and the graduates will produce a "portfolio" of products (both code components and documents) that can be shown and demonstrated to prospective employers.

## 5. Variants of Software Systems Engineering

The institutes that offer Software Systems Engineering programmes must choose between breadth and depth. They will have to balance teaching general software development principles against focussing on a particular class of applications. Each has both advantages and disadvantages.

- The graduates of a general Software Systems Engineering programme will be able to work in many application areas, but may lack familiarity with particular areas of knowledge that are relevant for some applications.
- Graduates of a specialized programme will be highly qualified for work in their own application area but may require extra education and effort if they move to a different class of applications.
- Graduates of a specialized area are usually well prepared to work with non-software professionals in that application area but graduates of a general Software

[7] Typical roles would be requirements analyst, modeller interface designer, implementor, tester, module user, documenter, etc. A student should be assigned several different roles but need not cover all possible roles.

Systems Engineering programme are often not prepared to discuss technical details with their non-software colleagues.

Whatever choices an institution makes, its programme should be clearly described so that prospective students and employers can make informed choices.

Below we illustrate the breadth of the Software Systems Engineering field by listing a few possible programmes (in alphabetical order). This list is far from complete; it is intended only to illustrate the broad range of choices available to educational institutions.

## 5.1. Communications system software engineering (CSSE)

The world has been transformed by the availability of systems that are capable of transmitting information over long distances at high speeds. Software is at the heart of telephone switching systems, mobile telephony, the internet, broadcasting systems, and local area networks. Software Systems Engineers who practice in this area require knowledge of how communications systems work, communication protocols, network interfaces, and applied information theory.

## 5.2. Information Systems Engineering (ISE)

Modern organizations depend on the rapid availability of information about their environment, employees, business processes, ongoing operations, and customers. They need systems that can support their work by finding, filtering, and presenting the right information to an employee, manager, or customer at the right time.

Software and data bases are critical technologies in building systems that provide an organization's need for information. Information systems engineering programmes should equip their graduates with methods, knowledge, and skills that will enable them to develop computer-based information systems that are tailored to the organizations they serve.

Information Systems Engineers require considerably more understanding of organizational structures, behaviour, decision making, and business processes, than would be needed by other types of Software Systems Engineers.

Section 6 of this chapter discusses ISE programmes in more detail.

## 5.3. Mechatronics engineering (software intensive engineering)

Software is now replacing analog technologies in many classical engineering applications. Software is critical in the control of manufacturing systems, transportation systems, robotics, navigation systems, aircraft design, weapons systems, etc.

Software Systems Engineers who will practice in these areas may require an understanding of many topics that classical engineers study, e.g., physics, differential and integral calculus, chemistry, thermodynamics, and materials science.

## 5.4. Software engineering (SE)

"Software Engineering" usually denotes a broad Software Systems Engineering programme that is intended to be application independent. SE programmes teach students about many types of software rather than specializing in a particular class of products. Because there are no application-specific requirements, it is possible to allow students to

go more deeply in specific areas than would be possible in an application-specific Software Systems Engineering programme. Some possible areas for additional depth would be:

- Computer science topics such as graphics, robotics, networking, or search algorithms,
- Software Systems Engineering topics such as quality assurance, security, interface design, or documentation,
- Mathematics topics such as graph theory, logic, queueing theory, differential equations, or statistics,
- Engineering topics such as thermodynamics, manufacturing processes, or control systems.

### 5.5. System-Software Engineering

All modern software is built using "system software" such as operating systems, device drivers, network interfaces, compilers, data base systems, and file systems. These products are often hardware dependent and have stringent reliability, security, speed and resource utilization requirements because all other software in the system depends on system software functioning correctly and efficiently. System-software engineers specialize in this type of product and will require specialized knowledge in concurrency, run-time error handling, resource allocation, system security, and hardware/software interfaces.

## 6. Additional learning requirements for Information Systems Engineering

For each of the variants of Software Systems Engineering, other than Software Engineering[8], it is necessary to formulate additional programme requirements that are specific to that (more specialized) discipline. As this committee evaluated several information systems engineering (ISE) programs, we have prepared a list of capabilities (and described one course) that we consider especially important for information systems engineers.

### 6.1. Organizations and management

Information systems engineers should have an understanding of organizational structures, organizational behaviour, business processes, decision making, and the business environment. Specific topics that need be covered include: human resource management, accounting, finance, marketing, operations and logistics management.

In short, without an understanding of how an organization works, possible alternative organizational structures, and the information needs of individuals in the organization, ISE graduates will find it difficult to design effective information systems.

[8] Software engineering is an exception because it is not specialized for a particular application domain.

## 6.2. System analysis

A problem is considered to be a *system problem* if the problem solver is given a collection of parts whose properties are assumed to be fully understood and asked to determine the emergent properties of the collection.

A critical systems problem for Information Systems Engineers is understanding an organization's need for information technology and systems, evaluating alternative solutions, and deciding on the best approach. An important tool for attacking this problem is known as "*business process modelling"*. Business process modelling helps the ISE to understand how the organization operates, and identify its information systems needs.

The analyst:

- creates models, often in the form of graphs, in which the nodes represent subdivisions of an organization or steps in its business processes,
- describes the behaviour of the organization's subdivisions, or steps in its processes, by simplified input/output relations, and
- derives the behaviour of the model.

A wide variety of charts with associated analytical methods have been proposed over the years; these can be useful to information systems engineers throughout their careers. There are many tools designed to help an analyst to create and use such charts. An Information System Engineer should understand the assumptions underlying each tool and be able to use that understanding to select the best tool(s) for each project.

In short, system problems arise in studying both the organizations that use information systems and the information systems themselves. Information systems engineers should learn a variety of methods for solving these problems.

## 6.3. Analyzing and organizing large amounts of data

Today's organizations have access to vast amounts of data; the individuals who make decisions for the organization are no longer able to process the data in its raw form to get the information that they need. The organizations use information systems to "mine" the available data and build a "web of data" (semantic web) that can be used to provide the organization with the information that it needs.

Two basic approaches to dealing with large amounts of data are filtering and aggregation. When filtering, one attempts to eliminate the data that are irrelevant to the person receiving the information. When aggregating, one computes functions of the data items to provide summaries that give the recipients useful information that can be derived from the data.

The classic problem of exploiting an organization's data resources has recently become a popular research field known as "Big Data". It is important for Information systems engineers to understand the work being done in this area.

In short, information systems engineers should understand methods of filtering and aggregating data that will help them to design information systems for their organization.

## 6.4. Information provenance and information validation

The information available to organizations comes from a broad variety of sources of varying trustworthiness; consequently, some data must be treated as lower quality than other data. Further, apparently similar information from different sources may not be defined or measured in the same way. Finally, information obtained in the past may no longer be valid when it is used.

In short, information systems engineers should learn how to keep track of the sources of the information that they process, how to assess the quality of information in their data bases and inputs, and how to deal with information that is not completely trustworthy.

## 6.5. Scientific decision making and risk analysis

Many advanced organizations make decisions based on quantitative models that provide an analytical approach to decision making and problem solving. These include operations research or optimization as well as mathematical methods of risk analysis. An organization's information systems may be expected to apply such methods in order to help managers to make decisions. Information systems engineers should have an understanding of quantitative, rule-based, and other decision making procedures so that they can incorporate them in the systems they build.

In short, an information systems engineer should have a basic background in what is often called "management science".

## 6.6. Managing IT systems in multidivisional organizations

Large organizations are often organized as a set of smaller organizations (divisions) that have a certain amount of autonomy. In the area of Information Technology this can bring numerous problems such as:

- Duplication: Several divisions may build systems that perform the same function.
- Incompatibility: It may be difficult for a system built within one division to exchange data with systems built by other divisions.
- Inconsistent Interfaces: Clients and suppliers may find it hard to deal with differences in the behaviour of IT systems in different divisions.
- Outsourcing policies: One division may perform a function "in-house" while others give contracts for those functions to external developers.
- Inconsistent standards: The IT field has many competing and overlapping standards. Each division might pick its own standards with the result that an organization will have conflicting policies.
- Security issues: One division may release information that another division treats as confidential or withhold information that another division should be able to access.

In short, Information systems engineers need to learn how to design and manage an "enterprise architecture" that allows both the individual divisions and the complete organization to function efficiently.

## 6.7. Overview of Information Systems Engineering

Early in an information systems engineering programme students should get an introductory overview of their chosen profession. It should discuss such topics as:

- Organizational structures and functions
- Roles played by information systems in an organization
- The functions of the main types of organizational information systems
- Identification of the potential users of an organization's information systems
- Identification of an organization's information requirements
- Identification of individual user's information needs
- Characteristics and value of information for its various users
- The organization of information
- Filtering and analyzing information relevant to an organization
- Exchanging information with users (user interface design)
- Integration of information systems that may have been separately developed.
- Decision making techniques (Management Science, Operations Research)

In short, beginning Information Systems Engineering students should be given an understanding of what they will have to learn to become professional Information Systems Engineers.

## 7. About curriculum design

Because there are many useful variants of Software Systems Engineering, we suggest that institutions carefully discuss which variants (see Section 5 of this chapter) are both (a) suitable for their students and (b) match the strengths of their faculty members. There are many useful variants and it would not be good if all institutions offered the same variant. The variant(s) should be clearly labelled and described to help potential students to make an informed choice.

Believing that there are many valid ways to teach students who want to be Software Systems Engineers, and many ways to organize the material into courses, the committee suggests that the capabilities listed in Sections 3 and 6 of this chapter be used as a checklist when designing or revising a curriculum. Institutions should look at each of the capabilities listed and ask, "How will our students learn to do that?"

Many methods have been proposed for performing each of the tasks mentioned in this report; curriculum designers will have to decide which methods to teach.

There are also many ways to teach each method. Further, some methods can be introduced or mentioned in courses but students can best obtain experience in using them when they are on the job. Curriculum designers must decide how best to use the limited time available to them.

This committee has focused on those aspects of Software Systems Engineering that differentiate it from Computer Science and programming. In addition, we discussed the

differences between Information Systems Engineering and the other Software Systems Engineering disciplines. We have emphasized the most basic (core) capabilities. Obviously, the relevant aspects of Computer Science and programming and some current issues must also be included in these programmes. Students need both the basic capabilities and the ability to “hit the ground running” when they enter industry.

Each institution will have to decide how to help their students to obtain the necessary capabilities, how to package the concepts and techniques into courses and projects, and how much attention each capability will get in their programmes.

## 8. Summary

Brian Randell, Fred Brooks, and others clearly identified the key differences between Computer Science, a research field, and Software Systems Engineering, a class of professions. Unfortunately, in the (more than) four decades that have passed since then, we have not seen enough progress in the professionalism and discipline of software developers. There is a great need for programmes that prepare people for a profession as a Software Systems Engineer.

It is important that programmes that are identified as Software Systems Engineering programmes help their students to acquire the capabilities discussed in Section 3 of this chapter. In addition, Information Systems Engineering programmes should include courses that help their students to acquire the capabilities discussed in Section 6 of this chapter.

Finally, Software Systems Engineering programmes should not be viewed as extended Computer Science programmes or as advanced programming programmes. Professional software system development is more than programming and requires many capabilities that are not taught in Computer Science programmes. Further, some parts of Computer Science are not essential for Software Systems Engineers; curriculum designers must exercise judgement about teaching material not included in the core that we have outlined.

## 9. References


1. Brooks, F.P., *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley, Reading, MA (1995), Second Edition.
2. Buxton, J. N. and Randell, B. (Eds.): *Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee,* Rome, Italy, 27 to 31 October 1969, Brussels, Scientific Affairs Division, NATO, April 1970, 164 p.
3. Naur, P. and Randell, B. (Eds.): Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7 to 11 October 1968, Brussels, Scientific Affairs Division, NATO, January 1969, 231 p.

# Chapter 5: Evaluation of the Software Engineering Study Programme at The Technion

*This Report relates to the situation current at the time of the visit to the institution, and does not take account of any subsequent changes. The Report records the conclusions reached by the Evaluation Committee based on the documentation provided by the institution, information gained through interviews, discussion and observation as well as other information available to the Committee.*

## 1. Organizational Structure

### 1.1. Observation and findings

At the Technion, the Software Engineering programme is a track taught by the Computer Science Faculty; it is run by a six person committee. This organizational structure is working well for the present programme. If the Faculty were to introduce a Software Engineering programme along the lines suggested in Chapter 4, a separate department could have some advantages.

### 1.2. Recommendations:

#### 1.2.1. Short term/immediate (~ within 1 year)

The Technion and the CS Faculty should consider whether or not to transform its "track" to a professional[9] programme that teaches the multi-person construction of multi-version programs as discussed in Chapter 4 of this report.

#### 1.2.2. Intermediate (~ within 2-3 year)

If the Technion does choose to have a programme like the ones described in Chapter 4 of this report, a separate department, staffed with faculty members who are dedicated to Software Engineering, should be formed. Technion students would then have a clear choice that they could make early in their studies. The programme and courses that they take could be optimized for Software Engineering.

#### 1.2.3. Long term (until the next cycle of evaluation)

A Department of Software Engineering should be staffed to teach a programme that has the aims described in Chapter 4 of this report.

## 2. Mission and Goals

### 2.1. Observation and findings

We found the mission statement that was provided to be very generic and not sufficiently specific to the field of Software Engineering. We wondered if there was significance in its use of the word "train" rather than "educate". In our opinion, university programmes should educate, not train. The mission statement stresses that the students should gain extensive experience in the production of software but the committee questions whether this goal is achieved; the stress of the track is on programming rather than the full process of building software.

---

[9] We use the term "Professional" to denote the style of education that is commonly used in Schools of Law, Medicine, Architecture, and Engineering programmes. Usually such programmes are designed to satisfy professional licensing requirements and are much more restrictive than programmes for Liberal Arts and Science degrees.

### 2.2. Recommendations

#### 2.2.1. Short term/immediate (~ within 1 year)

The Faculty should reflect on the alternatives available and write a mission statement that states the goals of the programme clearly.

#### 2.2.2. Intermediate (~ within 2-3 year)

The Faculty should use the revised mission statement to reconsider the educational programme and design a new programme.

#### 2.2.3. Long term (until the next cycle of evaluation)

The Department of Software Engineering should be teaching the new programme.

## 3. Study Programmes

### 3.1. Observation and findings

1. The programme does not fully correspond to the mission statement. The programme is a track in CS. The content is CS with some extra material on programming.
2. There is no course that offers an overall introduction to Software Engineering.
3. The SE methods course is badly overloaded. There is insufficient time to learn to use the methods.
4. There is no course on Human-Computer Interfaces (or Human-Computer Interaction).
5. Computer Security does not get enough attention.
6. The capstone project is a weak point of programme.
7. Software Quality Assurance (SQA) is not adequately discussed in the programme.
8. No course on databases is required of all students in the SE track. There should be one because of the importance of databases in many software applications.
9. Courses on computer system security, project management, and software design should be required of all SE graduates. The organization of this material into courses could be reconsidered. Some topics can be distributed over several courses. The design of Software Engineering programmes is discussed in greater depth in Chapter 4 of this report.

### 3.2. Recommendations

#### 3.2.1. Short term/immediate (~ within 1 year)

1. A curriculum committee, possibly including a few external experts whose area of interest is in the core areas of software engineering, should formulate an appropriate mission statement and begin to specify a programme that is designed for Technion students who want to become professional software designers and developers.
2. A curriculum structure that encourages students to decide between CS, ISE, and SE early in their studies would make it easier to offer better Engineering programmes.

### 3.2.2. Intermediate term (~ within 2-3 year)

1. Begin teaching an early course that offers an introduction to Software Engineering so that students have an overview of their chosen field and an understanding of the way that they will be able to use what they are learning.

2. More time should be devoted to methods for improving the multi-person construction of multi-version programs as described in Chapter 4. This material cannot be properly covered in a single “Software Methods “ course.

3. Computer Security should be discussed in many courses because it is not possible to add security as a separate feature. Every design decision may have security implications.

4. The Faculty should consider a US/Canadian style “Cooperative Education” (Co-Op) programme in which students get jobs in industry with a project that is supervised by both a company advisor and a Technion faculty advisor. In such programmes the company advisor is responsible for the day-to-day supervision, but the academic advisor makes sure that there is an opportunity for the student to learn by doing what has been taught in the student’s educational programme.

5. The Faculty should redesign its approach to projects.
    - All projects should have milestones with firmly specified deliverables including professional documentation for both maintainers and users.
    - Projects should include feedback from users and customer(s) and someone taking the role of an external (outside the team) maintainer.
    - In an SE project, all documents should conform to specified documentation standards, be dated, and properly identified.
    - Minutes should be kept during all project meetings and retained as part of the project’s formal documentation.
    - Projects should be retained (in electronic form) for future use by the Faculty; students should be advised to include them in a “portfolio” for future job interviews.
    - There should be firm course prerequisites for each project. The stress of each project should be on learning how to apply previously taught material. No new fundamental material should come up in a projects course.
    - There should be a project supervisory committee that works to assure consistency from year to year; it must meet to approve projects before they are given to students.
    - See Chapter 4 for a more detailed discussion of projects.

6. The faculty should design a course prerequisite structure to assure that all necessary courses are taken and taken in the right order.

### 3.2.3. Long term (until the next cycle of evaluation)

None.

## 4. Human Resources / Faculty

### 4.1. Observation and findings

The committee was positively impressed by the quality of the Technion's CS faculty. Generally, they can be described as "great people". However, although many are doing programming, there are not enough faculty members researching in the core areas of SE and not enough who have a combination of industrial development experience and research in the core areas of SE. Although the need for hiring in this area is appreciated by the faculty, recent hires have been in other areas.

The faculty take their teaching obligations seriously and are appreciated by students because they are accessible and supportive of students. The students made this very clear.

### 4.2. Recommendations

#### 4.2.1. Short term/immediate (~ within 1 year)

A steering committee, focussed on this programme, including some industrial and foreign experts, could contribute positively to the hiring and programme development. A faculty-wide industrial committee could review any hiring and programme proposals but primary control should remain with a committee dedicated to the SE programme.

#### 4.2.2. Intermediate term (~ within 2-3 year)

Since the faculty has chosen to design and teach an SE programme, we find a need to extend its faculty by hiring people whose research is centred on SE issues. Such people should be in charge of the programme. Taking them from industry, or "borrowing" from industry could be one way to get around the shortage of people. It is in the interest of companies who employ people in Israel to support a leading institution like the Technion.

#### 4.2.3. Long term (until the next cycle of evaluation)

None

## 5. Students

### 5.1. Observation and findings

The students that we met were excellent students, well prepared, smart, vocal, focussed and enthusiastic. In the past they have been well received by industry. The students are a major strength of the programme.

The students that we met had no complaints (except for one anonymous complaint about computer network performance).

### 5.2. Recommendations

#### 5.2.1. Short term/immediate (~ within 1 year)

None

#### 5.2.2. Intermediate term (~ within 2-3 years)

None

#### 5.2.3. Long term (until the next cycle of evaluation)

None

## 6. Teaching and Learning Outcomes

### 6.1. Observation and findings

With such good students it is hard to go wrong. However, we noted the lack of formal studies of the extent to which the learning goals are being achieved.

### 6.2. Recommendations

#### 6.2.1. Short term/immediate (~ within 1 year)

Review the programme in the light of comments in this report and identify the desired learning outcomes more clearly.

#### 6.2.2. Intermediate term:

Prepare descriptions of the learning outcomes in a form that specifies how a reviewer could measure the extent to which they are being achieved.

#### 6.2.3. Long term (until the next cycle of evaluation)

None

## 7. Research

### 7.1. Observation and findings

The faculty members are doing excellent and interesting research but they are not doing research in areas that the committee considered to be core SE areas.

### 7.2. Recommendations

#### 7.2.1. Short term/immediate (~ within 1 year)

Identify a list of research areas that should be the highest priority for the hiring process. Resist the temptation to hire the best available applicants even if those applicants do not research in the areas identified; instead focus on filling posts in the priority areas.

#### 7.2.2. Intermediate term (~ within 2-3 years)

Make area coverage an absolute priority.

Where necessary, seek the support of visiting faculty from industry.

Invite visitors in the priority areas and institute a “grow your own” programme by helping young Faculty to change to the identified areas.

#### 7.2.3. Long term (until the next cycle of evaluation)

None

## 8. Infrastructure

### 8.1. Observation and findings

The computing infrastructure was shown to us when lightly loaded; under those conditions it looked adequate. We did not investigate this issue closely. Building, classrooms, library and other such facilities were more than adequate.

### 8.2. Recommendations

#### 8.2.1. Short term/immediate

None

#### 8.2.2. Intermediate term (~ within 2-3 year)

None

#### 8.2.3. Long term (until the next cycle of evaluation)

None

## 9. Self-Evaluation Process

### 9.1. Observation and findings

The committee found that the self-evaluation report was hard to use for our purposes. Too much of it was devoted to self-promotion (often in the form of pictures). It was difficult to find the information that was relevant to our task. We wanted to see more of the information that was given to students as well as more information on course content.

The process description in the report was too generic to be helpful. We do not feel that this problem is specifically a Technion problem.

### 9.2. Recommendations

#### 9.2.1. Short term/immediate (~ within 1 year)

None

#### 9.2.2. Intermediate term (~ within 2-3 years)

None

#### 9.2.3. Long term (until the next cycle of evaluation)

Write a shorter self-evaluation report but make sure that it focusses on presenting facts about the programme rather than presenting positive (but often irrelevant) information. Include the information that is provided to students to help them to choose the right programme. This will help future committees to see whether or not the students are getting the programme that they were promised.

## Chapter 6: Summary of Recommendations and Timetable

### Short term [~ within 1 year]:

The Technion should consider whether or not it wants to transform its "track" to a professional[10] programme that teaches the multi-person construction of multi-version programs as discussed in Chapter 4 of this report.

A curriculum committee, possibly including a few external experts whose area of interest is in the core areas of software engineering, should formulate an appropriate mission statement and begin to specify a programme that is designed for Technion students who want to become professional software designers and developers.

A curriculum structure that encouraged students to decide between CS, ISE, and SE early in their studies would make it easier to offer better Engineering programmes.

### Intermediate term [~ within 2-3 years]:

The Faculty should produce a mission statement that makes a clear statement of the goals of the programme.

A separate department, staffed with faculty members who are dedicated to Software Engineering, should be formed and staffed with responsibility for teaching the newly designed programme gradually transferred to the new department.

The programme should include an early course that offers an introduction to Software Engineering so that students have an overview of their chosen field and an understanding of the way that they will be able to use what they are learning.

Several courses should be devoted to methods for improving the multi-person construction of multi-version programs as described in Chapter 4. This material cannot be properly covered in a single "Software Methods " course.

In addition to a course on computer security, security should be discussed in many courses because every design decision may have security implications.

The department should consider a US/Canadian style "Cooperative Education" (Co-Op) programme in which students get jobs in industry with a project that is supervised by both a company advisor and a Technion faculty advisor. In such programmes the company advisor is responsible for the day-to-day supervision, but the academic advisor makes sure that there is an opportunity for the student to learn by doing what has been taught in the student's educational programme.

The department should redesign its approach to projects.

- All projects should have milestones with firmly specified deliverables including professional documentation for both maintainers and users.
- Projects should include feedback from users and customer(s) and someone taking the role of an external (outside the team) maintainer.

---

[10] We use the term "Professional" to denote the style of education that is commonly used in Schools of Law, Medicine, Architecture, and Engineering programmes.

- In an SE project, all documents should conform to specified documentation standards, be dated, and properly identified.
- Minutes should be kept during all project meetings and retained as part of the project's formal documentation.
- Projects should be retained (in electronic form) for future use by the Faculty; students should be advised to include them in a "portfolio" for future job interviews.
- There should be firm course prerequisites for each project. The stress of each project should be on learning how to apply previously taught material.
- There should be a project supervisory committee that works to assure consistency from year to year; it must meet to approve projects before they are given to students.
- See Chapter 4 for a more detailed discussion of projects.

The faculty should specify course prerequisites to assure that all necessary courses are taken and taken in the right order.

Prepare descriptions of the learning outcomes in a form that specifies how a reviewer could measure the extent to which they are being achieved.

Identify a list of research areas that should be the highest priority for the hiring process. Resist the temptation to hire the best available applicants even if those applicants do not research in the areas identified; instead focus on filling posts in the priority areas.

Make area coverage an absolute priority.

Where necessary, ask for the support of visiting faculty from industry. Invite visitors in the priority areas, and institute a "grow your own" programme by helping young Faculty to change to the identified areas.

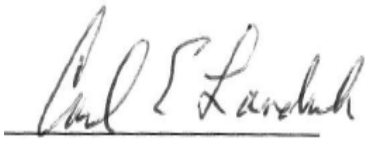
### Long term [until the next cycle of evaluation]:

There should be a Department of Software Engineering that is staffed to teach a programme that has the aims described in Chapter 4 of this report. The Department of Software Engineering should be teaching the new programme.

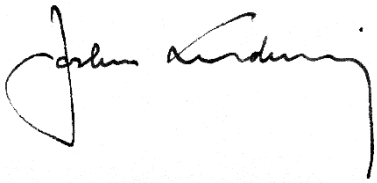
Write a shorter self-evaluation report but make sure that it focusses on presenting facts about the programme rather than presenting positive (but often irrelevant) information. The report should include the information that is provided to students to help them to choose the right programme.

**Signed by:**

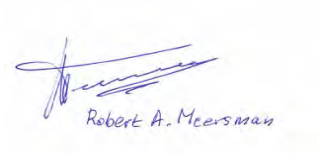
Prof. Dave Parnas



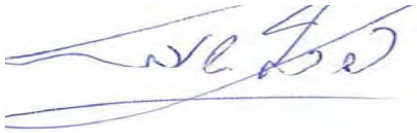
Prof. Carl Landwehr



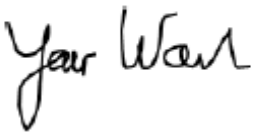
Prof. Jochen Ludewig




Prof. Robert Meersman



Prof. Peretz Shoval



Prof. Prof. Yair Wand

Prof. Elaine Weyuker