

CS1Ah Lecture Note 1

Science and engineering

This course provides an introduction to computer science and to software engineering. Are these not the same thing? One of the central concerns of computer science as a discipline is the development of techniques for the reliable production of software systems. Applying proven techniques in producing reliable software systems is one of the primary activities of a software engineer.

The two subjects are certainly related but they differ in their methods and practices. In computer science it is common to make mathematical models of computer systems, programs and programming languages. Computer scientists then work with these models, investigating them and discovering their properties and potentials. In this way, they may indirectly uncover facts about the systems, programs and languages from which the models provide an abstraction. The hope is that the abstraction process generalises some of the results which will be obtained, so that they may apply to similar systems, under similar circumstances.

In software engineering it is more common to work directly with systems, programs and languages. The necessarily complex nature of genuine artifacts of these kinds poses a formidable challenge to a software engineer. Using advanced tools for system design, well-tested techniques of software development and reliable, well-structured programming languages can help to meet these challenges. Some of these tools, techniques and languages will have been influenced by the results of computer science research but a software engineer is not merely a user of the products of computer scientists. Software engineers must also innovate and be creative but they also have the added responsibility of producing safe, well-engineered products which will be used in commerce, or industry or in the home.

1.1 Connecting the science and the engineering

At its best, computer science can be seen as an experimental science. A computer scientist develops a hypothesis about how computer systems or programming languages can be improved. This hypothesis can be investigated via small-scale laboratory experiments which might involve the creation of a prototype compiler for a programming language or a prototype tool for supporting the software development process. If these laboratory experiments prove to be successful then the results can be announced to the scientific community and to interested representatives of the software engineering

profession. If the initial laboratory investigation has proven to be persuasive then a larger-scale experiment of a similar kind might be conducted to investigate whether or not the potential benefits can be delivered when the experiment is scaled up. If the outcome of that experiment is successful, and other circumstances are also favourable, then the method might come into common use and be accepted as good practice within the software engineering profession.

Here is an example of this effect at work. In March 1968, the Association for Computing Machinery published a two-page long article by a Dutch computer scientist, Edsger W. Dijkstra.¹ The title of the article was *Go To Statement Considered Harmful*. (Reprinted on the Web at www.acm.org/classics/oct95.) Persuaded by the article, the group of programmers developing the UNIX operating system in the early 1970s at the University of California, Berkeley decided to write the entire system *without* using the `goto` statement. Persuaded by this experiment, use of the `goto` statement was viewed as bad practice within the software engineering profession. A modern programming language such as Java does not even have a `goto` statement.

The previous example might suggest that computer science is a sedate subject where effective ideas take decades to be realised. That would be misleading so another example might help to correct it. To take a more recent example, the Freenet large-scale *peer-to-peer* network pools the resources of a group of computers distributed around the world to form a massive virtual information store which is open to anyone to freely publish or view information of all kinds. Peer-to-peer is a radical new communications technology which is being viewed as “re-wiring” the Internet through software because it changes the fundamental philosophy underpinning the structure of Internet communications. Freenet was developed by a University of Edinburgh undergraduate student, Ian Clarke, for his final-year dissertation for the degree of Artificial Intelligence and Computer Science here in 1999. In the following year a paper on Freenet was published and a chapter on it appeared in the book “Peer-to-Peer: Harnessing the Power of Disruptive Technologies” published at the start of 2001. Work on Freenet continues, including featuring at an international conference in November last year. With this, in three years, it has gone from a student project to commanding world attention. Scientific progress, as with everything else, happens faster in the Internet age.

1.2 Separating the science and the engineering

Since progress in computer science stimulates progress in software engineering, and conversely, what is the difference between the two subjects? Primarily, computer science favours a blend of abstraction, general applicability, precise definition, and unapologetic, sometimes gratuitous, use of mathematics. Software engineering favours concreteness, specific applicability, characterisation, and restrained, sometimes begrudging use of mathematics.

One freedom which computer science enjoys is that it can commence the scientific analysis of computer systems before they are constructed. Some of the most important theorems used by computer scientists were developed by Alan Turing before the first computer was ever built. The same advanced position is also enjoyed by computer scientists reasoning about the next generation of not-yet-built quantum computers.

¹Sadly, after a lifetime of contribution to his science, Dijkstra passed away in August 2002.

One freedom which software engineering enjoys over computer science is the ability to consider problems and topics for which mathematical techniques are not well-suited. This includes the modelling of human organisational behaviour in the software development process and the creation of usable user interfaces for computer programs. Some of the insights which come from software engineering are distilled as *patterns*, structured and general descriptions of perceived good practice, which cannot usefully be described in mathematical terms.

To give an example of a subject which is of great interest to computer scientists but seemingly of little interest to software engineers, we could consider *parsing*. This is the process of checking the grammatical correctness of a computer program, with respect to a relatively simple grammar. There are literally thousands of academic papers on this subject and a widely-understood body of techniques which can be used in attacking parsing problems. Parsing also arises in a great number of applications and yet Java, the language of choice for the working software engineer, provides no special support for writing parsers.

To give an example of a subject which is of great interest to software engineers but seemingly of little interest to computer scientists, we could consider *version control systems*. A version control system manages access to the texts of computer programs, allowing software engineers to acquire copies of programs rather like borrowing books from a library. They can then work on the program and return it to allow another engineer to develop it further. The version control system asks them to add a note explaining what changes they have made, and why. This is a very useful tool for sharing programs between the members of a software development team but such programs have received relatively little study from computer scientists.

1.3 The role of modelling

One common point of comparison between computer science and software engineering is found in the role of *modelling* in the two subjects. Both computer scientists and software engineers make models of more complex systems and work with them, reasoning about the model in order to obtain insights into the more complex system which it represents. The creation and use of models serves to abstract away from unimportant details of the system; this is one reason why a computer scientist would make a model of a programming language. In the case of a software engineer it might be that the actual system under study is too dangerous or too valuable to experiment with directly. For this reason it is a common practice to have a smaller, isolated version of the *live* system known as the *production* system. Development work and experimentation takes place on the production system before it is moved across to the live system. The production system serves as a model of the live system.

1.4 The role of languages

Another common point of comparison between computer science and software engineering is found in the role of *languages* in the two subjects. Both computer scientists and software engineers work with a range of formal languages and informal notations, used for a variety of different purposes. Both formal and informal languages are useful. Sometimes it is necessary to consider fine details about the system (and so a formal

language is the right tool) and sometimes one wants to take a step back and consider the big picture (an informal language is more helpful here).

One way of passing on the benefit of scientific progress from one generation to another is by designing programming languages which reflect the current best practice in software development. The Java programming language embraces many good insights. It supports the most commonly used programming idiom; *object-oriented* programming. It has a *strong type system* which checks for certain kinds of errors in programs before they are ever executed. It does automatically for the programmer some tasks with other programming languages do not attempt; *automatic memory management* is an example of this.

However, it is inherent to the nature of programming languages that they are usually subject to revisions and extensions which are made in order to try to better support the working software engineer by turning the programming language into a more productive working tool. Programming languages can be surprisingly long-lived. The Fortran language is nearly five decades old (its major revisions were FORTRAN 66, 77, 90 and 95). It is still in widespread use and work is in progress on a new Fortran 2002 draft standard. The Java programming language is in comparison very new; it was released in the mid-1990s. However it has been revised more vigorously than any programming language before it. Because it is revised roughly every nine months Java releases are numbered (Java 1.0, 1.1, 1.2, ...). We use Java 1.4 on this course.

1.5 The importance of *correctness*

Perhaps one of the most puzzling aspects of computer science for those beginning studying the subject is “why do computer scientists place so much emphasis on the subject of the *correctness* of computer systems?” Relatively speaking, compared to other industries, the software industry has a highly commendable safety record. The number of times where loss of human life was caused by an error in a computer program is remarkably low considering the widespread use of software in medical equipment, consumer appliances, transportation systems and the like. Perhaps the motivation for this concern is to preserve this commendable safety record or it just comes from taking simple pride in doing a job well. It seems more likely though that computer scientists are attracted to this question by the richness of the technical questions which arise when determining effectively whether or not a software system should be labelled as correct. For software engineers the question has a straightforward answer. Software systems are usually commissioned by a customer who develops a contract which the software engineer agrees to meet. Such contracts have a legally binding status and failure to meet the terms of the contract could result in non-payment for the work or punitive legal action.

The conclusion that could be drawn from all of the above is that computer science and software engineering are both earnest, useful endeavours. Striking a balance in the appreciation of both subjects leaves neither unfairly ignored. Individuals can favour one or the other according to their personal taste.

*Lecture note by Stephen Gilmore.
David Aspinall, 2002/09/15 17:05:07.*