

Software Needs Engineering

— a position paper —

Jane B. Grimson

Department of Computer Science
Trinity College
Dublin 2
Ireland
+353-1-608 1780
Jane.Grimson@cs.tcd.ie

Hans-Jürgen Kugler

Q-Labs
5 Kendalstown Rise
Delgany, Co. Wicklow
Ireland
+353-1-287 1049
HansJurgen.Kugler@q-labs.com

ABSTRACT

When the general press refers to ‘software’ in its headlines, then this is often not to relate a success story, but to expand on yet another ‘software-risk-turned-problem-story.’

For many people the term ‘software’ evokes the image of an application package running either on a PC or some similar stand-alone usage. Over 70% of all software, however, are not developed in the traditional software houses as part of the creation of such packages. Much of this software comes in the form of products and services that end users would not readily associate with software. These can be complex systems with crucial connections made through software, such as telecommunications or banking systems, or the logistics systems of airports. Or these can be end-user products with software embedded, ranging from battery management systems in electric shavers, over mobile phones to engine management and safety systems in cars. e-Commerce systems fall into this category, too.

Yes, there is software that works reliably and as expected, and there are professional approaches to create such products — one can *engineer* software, in the right environment, with the right people.

Keywords

software, engineering, profession

1 CASE 1: “MISSION COMPLETED, BAGGAGE DESTROYED.”

Many travellers have experienced the chaos airports and their logistics can inflict on their personal plans when it

comes to baggage ‘handling.’ However disastrous such an experience may have been, it is most certainly exceeded by what the system of Denver International Airport was capable of, or, was *not* capable of. The project to build an automatic baggage handling system was added in 1992 as an additional aspect of building the new Denver International Airport (DIA), which is twice the size of Manhattan and 10 times London’s Heathrow airport. The DIA baggage system is a subterranean baggage handling system consisting of 21 miles of steel track connecting all the terminals facilities designed to house 20 airlines. Baggage is automatically routed on these tracks in 4,000 independent carts, controlled by more than 100 networked computers and countless sensors and radio communicators. At the heart of the operations is a purpose built software system monitoring and controlling the movement of the baggage in the carts.

Unfortunately industry experience shows that such projects to develop and install large software systems are on average 50% late, with corresponding over-expenditure and serious “teething problems” even after delivery, and the DIA baggage system was no exception. Problems with the software system delayed the opening of the airport from October 1993 in various stages until February 1995 — in the meantime a conventional system for baggage handling had been constructed as backup. As late as 1996, with the automated system in operation, there were still considerable problems to be resolved.

What went wrong?

Whilst there were some mechanical problems and some problems with sensors, the real problems were with the control software of the baggage system. These problems were not simple programming defects that could be remedied in a rather straightforward manner; no — these problems were “built in” right from the start through inadequate handling of the requirements the software system was supposed to fulfil. In particular, permitting changes to the requirements and the specification in a less than controlled

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE 2000, Limerick, Ireland

© ACM 23000 1-58113-206-9/00/06 ...\$5.00

manner compounded this. It is not possible to attribute the failure of a software project to just one single cause, but lack of discipline in software management processes permitted the risks to grow.

2 SOFTWARE SYSTEM FAILURES — MAJOR CAUSES

Robert Glass [1] categorises major causes of software system failures as:

1. project objectives not fully specified
2. bad planning and estimating
3. technology new to the organisation
4. inadequate (or missing) project management method
5. insufficient senior staff on the team
6. poor performance by suppliers of hardware/software
7. (other) performance or efficiency problems

None of the above categories is really technological at heart; they are all related to software management, i.e. the lack of the application of proper engineering management to the software part of a larger project, or the lack of risk management in technology adoption.

3 CASE 2: ‘SLAIN BY THE SAVIOUR’

The (unsuspecting) user does not interact with software only through the result of such large project undertakings. Software controls many functions of a modern car. Many of the recent advances in road safety and fuel economy of cars depend on fairly complex safety components and motor management systems, which, not surprisingly, are essentially software systems. Modern cars have nearly as much software under the bonnet as PCs used to have in their operating systems. The users’ expectations are generally higher, though — no motorist wants to be forced to reboot the engine of a car several times a day.

Safety devices, such as anti-lock brakes and airbags are expected to be fail-safe. Last year the German magazine ‘Der Spiegel’ reported an accident, in which a baby sitting in a rear facing baby seat mounted to the front passenger seat of a car was killed by the impact of the deploying airbag in an oncoming traffic collision. This happened in spite of the fact that the airbag had been disabled previously at a certified garage.

Experts from the Technical University of Munich suspect a systems engineering fault as the likely causal factor. Deactivation of the airbag in this make of car was¹ a software-controlled function. As physical circumstances in

¹ This has subsequently be changed to physical disconnection of power.

a car (temperature variations, moisture, and vibrations) form a fairly hostile environment for the air bag control hardware, the software system running inside the control performs ongoing self-checks. If an error has been detected, the current software settings, including the data responsible for the deactivation, will be replaced by backup software read out of a read-only memory. This backup software only knows a simple set of rules; namely that all air bags in the car will be deployed upon impact. Evidently, the backup software is unaware of a previous software-controlled deactivation. Other manufacturers physically disconnect the air bag from the board electronics for deactivation.

Again, this is not a technological problem with software as such, but a problem of the relationship between the behaviour of the software and the behaviour of the wider system. This may be another problem of “project objectives not fully specified”, either initially or by failing to control a later modification of the system. Project management, team seniority or even team communications can also have played a role.

4 SOFTWARE NEEDS ENGINEERING DISCIPLINE

The dictionary definition of *engineering* is the application of scientific and mathematical principals towards practical ends. Thus engineering is concerned with creating cost-effective solutions to practical problems by applying scientific knowledge to building ‘things’ - or systems - in the service of mankind. Engineering is not just about solving problems. It is about solving problems subject to economic, environmental, social, and other constraints. It is about the economical use of all available resources including money. It deals with practical problems whose solutions matter to people outside the engineering domain. Engineering solves problems in a particular way by applying scientific and mathematical principles to analyse, construct and test artifacts or systems. Software Engineering falls into this generic description of what Engineering entails.

Software does not wear out in the physical sense but it does age. David Parnas one of the leading proponents of computing as an engineering discipline stated “*A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long term health of our products. Researchers and practitioners must change their perception of the problems of software development. Only then will Software Engineering deserve to be called Engineering*”. . [2].

One of the major problems with the development of software is that so much of it is built in an *ad hoc* manner from scratch rather than by assembling it from off-the-shelf components. Traditional engineering artifacts are almost always built from standardised components. Because of the lack of a model of the system and inadequate documentation it was felt that it was easier to re-invent the

wheel rather than try to understand the existing wheel and adapt it to fit the new bicycle. The failure to combine tried and tested components to build new systems means that each new system is built from scratch in a monolithic fashion so maintenance is a nightmare.

The greatest advantage of software is its flexibility, and this is also its greatest drawback. The potential flexibility of software systems is unfortunately often equated with “easy to change.” Just think of what side effect the change of a macro or even just a simple re-used formula in a spreadsheet may produce. Now imagine what effect such “changes-on-the-fly” under time pressure in a complex real-time system may produce. Software technology needs engineering discipline to be able to repeatedly and economically produce reliable software intensive products and services. The fact that software is intangible is probably the greatest obstacle in introducing engineering rigour in many software projects.

Does this mean that software that meets expectations and constraints is not affordable? No, rigorous engineering processes are feasible, and examples of what is considered industrial good practice in software engineering are being deployed in more and more organisations. One such approach is based on the Capability Maturity Model² (CMM) [3] of the Software Engineering Institute at Carnegie-Mellon University. This is an application of Crosby’s quality grid to software, with lots of feedback on good practices added from practical experience. The CMM describes an evolutionary approach for software developing organisations to increase their capabilities to profitably produce higher quality software. At one of the intermediate steps focus is laid on adopting and continuing with good engineering practices in requirements management, project planning, project tracking, software quality assurance, subcontract management and configuration management. The relationship to the above mentioned problem areas is obvious.

Improved industry standard software processes, together with a professional software engineering team, have been shown to deliver higher quality to the customer and higher profits to the developers. There is a need for the transfer of an organisational engineering culture.

5 DO WE NEED PROGRAMMERS OR SOFTWARE ENGINEERS?

Actually, we need both, but for different tasks. Just as in traditional engineering disciplines, there is a need for both the professional engineer and the technician. They are trained differently and fulfil different roles. The primary educational qualification for the professional engineer is normally an accredited 4 or 5 year degree programme,

² CMM is registered with the US patents and trademarks office.

followed by a period of practical experience and on-the-job training. The typical engineering degree programme is based on the *engineering science* approach under which students study the underlying mathematics and science (principally Physics and Chemistry) appropriate to their discipline in the early years, followed by engineering applications. Engineering technicians and technologists, on the other hand, receive a shorter and more practically focused education to sub-degree level. There is generally much less emphasis on the underlying science and mathematics. In terms of their respective roles, the professional engineer is usually responsible for the design of engineering artefacts and systems, as well as project management, while the technician is generally responsible for the “construction” and maintenance of artifacts and systems. There is – or should be – a direct parallel with the construction of software systems with the software engineer being responsible for the design and project management and the “technician” (programmer, analyst, tester, etc.) being responsible for implementation, maintenance etc.

6 CONCLUDING REMARKS

Large civil engineering structures have been built since before recorded history but it is only in the last 150 years or so that their design and construction has been based on theoretical understanding rather than intuition and accumulated experience. By that yardstick, the software industry has made rapid progress indeed, but there is no doubt that much more needs to be done to improve the quality and reliability of software.

The invention of the digital computer in the 1940’s resulted from the collaborative work of engineers, scientists and mathematicians. For some years afterwards, it was felt that this fledgling discipline of Computer Science might be reabsorbed into mathematics, engineering or physics [4]. In some ways these tensions still exist with heated debates on the nature of computing as a discipline. Although few people today would agree with Lord Goodman when he wrote in 1986 “*We are living in a computer-obsessed world. A man trained in computer science alone is by any definition an uneducated man; the enormous danger that looms over us is that we shall be satisfied to have a society trained to this limited degree and ignore or deride the necessity for the older notions of liberal education*” [5]. It is interesting to recall that the very same criticisms were leveled against universities which sought to introduce engineering degrees in the universities in the mid-19th century.

Steve McConnell recently posed the question “*should software development be engineering?*” [6]. The answer has to be a resounding “Yes!”. As computing has matured, it has reached a stage where there is an extensive body of underlying theory and principles, as well as a vast array of practical tools and techniques. This in no way devalues the

contribution of mathematics and science to the discipline nor does it detract from the increasingly important role of other disciplines in modern software development including, for example, psychology, cognitive science, economics, graphic design and sociology. Indeed if anything the multi-disciplinary nature of software development strengthens the argument that it is an engineering discipline.

Modelling and simulation tools should play as key a role in software engineering as in other engineering domains. There is a vast array of tools available today which allow software developers to build prototypes/models of their systems and test them before engineering the final product. Yet we continue to build systems like the Wright brothers built aeroplanes – push them over the cliff, watch them crash and then start all over again! Modelling and simulation tools enable developers to “test drive” a specification of their system.

As with traditional engineering disciplines, we must aim to enter the new millennium with clear definitions of the professional competencies and responsibilities required for those working in the software industry. Similarly we need to move towards a certification process for software engineers [7]. This will in turn ensure that we engineer the next generation of software systems to the highest standards of performance, safety and reliability that reflect the central and critical role which these systems increasingly play in modern society.

REFERENCES

1. Glass, R.L. *Software Runaways — Lessons Learned from Massive Software Project Failures*. Prentice Hall, PTR, NJ, 1998.
2. Parnas D, *Software aging*, Talk given at the Technische Hochschule, Vienna, June 16, 1995.
3. Paulk, M.C. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, Reading, MA, 1994.
4. Denning PJ, *Computing the profession*, Computer Science Dept, George Mason University, Fairfax, VA22030, August 1998.
5. Lord Goodman, No whispered enhancements for the don, *Observer*, 29 June, 1986.
6. McConnell S, *The Art, Science, and Engineering of Software Development*, *IEEE Software*, Jan-Feb 1998, 118-120.
7. Maginnis T. *Engineers don't build*. *IEEE Software*, January/February 2000, 34–39.