



WHY HAS SOFTWARE BEEN SO DIFFICULT TO WRITE WELL?

Software failures can, and have, caused substantial damage, and loss of life. Why is it so difficult to engineer software correctly? Can software ever be as safely and rigorously designed as the projects created by traditional engineering disciplines? President of Engineers Ireland, Dr Chris Horn Chartered Engineer explored these questions in his recent Presidential Address

The emergence of the Fortran programming language, created by John Backus in 1953, marked a key step in the emergence of software engineering, since, at last, programmes could be developed and moved from one computer to another without requiring re-writing. Today, there are myriad programming languages, probably running to several hundred, each specialised in its own way. However, it is interesting to note that whatever programming language you use, it is technically possible to re-implement your work in any other programming language.

Given that in theory any two programming languages can be implemented in each others terms, and given that so much software has accumulated over the years using different programming languages, then it should be possible for any of the newer languages to re-use software libraries built with any of the older ones, right?

In general, today's answer is 'yes'. Software tools and middleware have been developed which automatically convert the format of data and numbers, and the invocation conventions, of one programming language to another.

Accept no substitutes

So, you are probably thinking, software should therefore be built using standardised plug-in parts which are well-specified and well-understood. Surely the software industry has defined a set of small, indivisible and substitutable software components?

Software components are rarely substitutable: there are no widely accepted industry standard specifications for a basic, common, industry-wide set of components available from a range of suppliers. It is as if each component of a bridge were available only from a single manufacturer, and as if different components from different manufacturers very rarely simply join together. Programming is often largely a craft in which each software artifact is custom-designed and built, rather than an organisational system like mass production manufacturing.

Larry Constantine in the 1990s observed: "If it takes the typical programmer more than two minutes and 27 seconds to find something, they will conclude it does not exist and therefore will reinvent it."

Software error impact: Case Study 1

In late September 1983, the world almost came to an end in nuclear annihilation.

Because of software failure.

Lt. Col. Stanislav Petrov was duty officer at Serpukhov-15, an early warning satellite system post, just outside of Moscow in September 1983 during one of the most tense periods of the Cold War. Five Minuteman ballistic missiles had apparently been launched from the US and were speeding towards Russia.

Petrov had been warned that a US missile strike would be massive, an onslaught designed to overwhelm the Soviet system. Although trained to recommend an immediate counter-launch, he waived.

Less than five minutes after the alert began, Petrov told his superiors that the launch reports must be false. If the US wanted to start a war, he reasoned it would have launched more than five missiles.

His guess was right. Later, it transpired that the software in the Russian early warning system mistook sunlight reflecting off clouds for missile plumes.

Today, search engines help programmers find existing code – usually in less than two minutes 27 seconds. Each modern language has numerous re-useable components. But still re-using software in new situations is hard.

Classical works

In classical engineering, civil engineers not only study the great buildings, but also the great bridges and transport systems. Mechanical engineers learn of the great engine designs which advanced their profession. Electronics engineers study model circuits, by which their profession advanced.

Until the last 15 years or so, students of software engineering rarely studied the great works of their profession. The great works were in fact unavailable to be studied: software companies jealously guarded the software code of their products, only allowing a limited number of employees to see their inner workings.

But even when works of software are available to be studied, the sheer size of a software system can be daunting. The English translation of Tolstoy's *War and Peace* has about 43,000 lines. The full release of Debian version 4 for Linux has about 283 million lines of code, which is thus about 6,500 times as big as *War and Peace*.

Management

How can the construction of large software systems, running to millions of lines of code, employing many software developers, and costing potentially large sums of money, be managed?

As a manager, you can, of course, measure the lines of software code written so far, and thus how productive your software developers are. However, this is misleading: large cumbersome, bloated code may contain many lines, but a more concise, carefully-written version may perform better and often will be easier for others to understand. There are other metrics, such as how many features have been developed so far - but then features can vary tremendously in the scope and effort required to implement them. You can measure how many defects have been detected and repaired - but sometimes as the defect rate rises, paradoxically you may be closer to a finished project. It has proven difficult to identify a general set of software management metrics useful across a wide range of projects.

Testing

How do you know whether a programme is correct and works, after it has been written? How do you know that it does what it is supposed to do? The obvious way is simply to test it: give it data and inputs, and see whether it produces the correct behaviour and results. Usually there are a large, or even infinite, number of test scenarios, with only a finite amount of time and resources available to investigate them. The consequence is that it is impossible to test everything. Software engineers have frequently operated under instruction that their work be 'good enough' - that a software system has no major problems, and should be OK to use, even though it may not be error-free. But what is a definition of 'good enough'? Is it by consensus across the team? Is it when predetermined criteria are met? Should testing cease once the costs of testing (including salary costs) have begun to climb above what is acceptable?

Rather than testing for some subset of possible inputs and timing conditions, could the software test itself? Self-testing software is of course common, and it is also common for software engineers to ensure that their software verifies that certain design assertions are valid as the software progresses.

Software error impact: Case Study 2

In July 2007, an agency of the Irish Department of Health and Children abandoned the Personnel, Payroll and Related system for 120,000 health care workers nationwide. The system used the SAP human resources package, under a managed service agreement with IBM. The project rollout was managed by consulting firm Deloitte. It was originally budgeted at €9m in 1999. By the time it was abandoned, it had cost some €220m. In one incident with the system, one employee was overpaid by €1m as part of an electronic funds transfer error.



Software error impact: Case Study 3

In March 2008, Ryanair's web site underwent a massive upgrade and was shut off for an entire weekend. It then stalled under load once it was powered up.

However, then there is of course the issue of what should be done if the software discovers, while it is running, that a design assertion is actually invalid. If an assertion is suddenly found to be invalid, how should the system recover with the failure in its design ?

A physics of software

The biggest issue is that software components almost invariably have state: they can record values, and their outputs in the future can depend on what inputs they have seen in the past. The concept of time and state is intrinsic in many software components as they model the real world: not only, 'what is my bank balance now?', but also 'what was my bank balance two weeks ago, or at 4.30pm on November 15, 2005?' If you are a structural engineer, imagine working with say, a beam, whose deflection response could be a complex derivation of all previous loadings throughout its use; or whose deflection response now was some function of a load at an arbitrary specific date and time in the past, such as 4.30pm on November 15, 2005. Software mathematics and theory have yet to produce a tractable methodology for the physics of state and how past behaviour modifies current and future behaviour. As software engineers, we lack a tractable physics of software that allows our profession to reason and predict how complex assemblies of varieties of software components will interact

Software error impact: Case Study 4

In October 2008, the Aer Lingus web site crashed and was unavailable for a couple of hours, under the load created by the airline's first 'no fare' offer.




Geotextiles on location in Ireland



Fibertex F-1200M
Protection Function
Ballealy Landfill Site,
Fingal County Council



Fibertex F-2B
Separation function
N6 Ballinasloe,
Galway County Council



Fibertex F-4M Separation
and Filtration Functions
Equestrian Arena,
McKee Barracks, Dublin

Fibertex Geotextiles are used in a wide range of applications where the outstanding properties of the material are a natural choice for:

- Road Works
- Construction Works
- Ground Systems
- Drainage & Filtration Systems
- Hydraulic Works
- Waste Disposal Sites

Available from Irish Tar on 01 855 5117

Irish Tar & Bitumen Suppliers, Alexandra Road, Dublin 1. Tel: 01 855 5117. Fax: 01 855 4262 Email: info@irishtar.ie




and behave under all the potential operating conditions which they may encounter during their lifetime. Without such a physics, it appears difficult to:

- analytically evaluate alternative designs;
- understand our systems, how they actually operate and sometimes why they fail; and,
- provide design guidelines with confidence;

Until we have such a physics, software engineering may continue to be a creative undertaking based on heuristics and experience, rather than an applied science with a sound analytical base.

Software safety

I hope it is clear from the case studies (see panels) which I have given you that software failures can be dangerous. I hope too that I have given you a few insights about the software profession, and why there are deep issues about the safety of software. Safety is a critical ethos of engineering. It is Engineers Ireland’s view that all engineering projects, which may affect the health and safety of the public, or may damage property, should be certified by a professional engineer. Currently, the Irish engineering profession is weakly regulated, and compares unfavourably to certain other jurisdictions in which certification of all engineering works is legally required. Engineers Ireland is taking initiatives to safeguard the public against poor engineering judgement and unprofessional analysis. For software, it seems natural to expect similar regulation and certification of software systems. For example, the EU medical devices directive requires certification of devices for healthcare - not only of electrical

Software error impact: Case Study 5

Over nine hours, on the afternoon of January 15, 1990, 75m phone calls failed and 200,000 airline reservations were lost in the USA. The USA nationwide AT&T telephone network collapsed when one single line of a programme in a complex software programme was wrong.

switches, plugs, cables, electromechanical devices and so on, but also of the software embedded in intelligent medical devices. Certifying that an engineering work is safe has reputational, fiscal, and ethical liabilities. Certification that a software system is safe raises deep issues about the nature of software itself. This is a global challenge, but also an opportunity for smart and innovative engineers. Those who can reflect sufficiently deeply on the nature of software itself, and derive a solid and pragmatic framework for ensuring that software can be safely designed so it can be certified as free from risk, will benefit mankind. Φ

This is an abridged version of Dr Horn’s presentation which can be viewed in full as a webcast at www.engineersireland.ie

Supported by:

Facilities Management Ireland

Ireland’s Largest FM Event

Combining FREE to attend conference sessions delivered by leading industry organisations, together with a major international exhibition, Facilities Management Ireland is your opportunity to access Ireland’s largest gathering of FM Professionals, the latest information, best practices, specialist suppliers and solutions that can benefit you, whatever the size of your organisation.

For full event details and online registration visit www.fmiireland.com.

Conference & Exhibition
23-24 March 2010. RDS, Dublin

Security Solutions

Energy & Sustainability

Building Services

Fire & Safety

Workspace & Environment