SOFTNTSYS

“Engineering” Software Systems

23 March 2021

“Software Development” ≠ “Software Engineering”

There has always been some debate about whether Software Development is really 'Engineering'.

In this paper, I discuss why ‘engineering’ should be a serious topic for software development (especially for critical software such as control systems) and the implications for the future of software development as good engineers.

My views are built on a career of designing, building or advising on control systems for equipment that can adversely affect human lives if it behaves incorrectly. These effects are frequently a result of the power or energy of the system involved to inflict damage; the safety element of human proximity to machines; or even environmental, societal or political damage that mis-operation might cause.

At the outset, I consider myself accepting of, although not content with, poorly designed software systems whose malfunctions are purely 'inconvenient'. I accept that they may be developed (designed and tested) to a lesser standard. I remain suspicious that, in those cases, little 'engineering' is involved (e.g. in websites or isolated informational systems).

However as more systems become interconnected, I feel concerned for the behaviour of larger systems to the consequential failure of such poorly thought through component systems of that larger system.

What does “Engineering” mean?

For me the process of engineering (a product, or service) is that intellectual sequence of steps at deriving the 'best' solution to a problem.

The derivation of 'best' is actually quite difficult, as it typically stems from a trade-off of lots of functional attributes (e.g. safety, security, availability, reliability etc) and non-functional attributes (e.g. weight, size, material cost, cost of engineering, cost of maintenance or ownership).

The stages of the engineering process are typified by:

- Understanding the problem
- Considering potential solutions
- Analysing the efficacy of the solutions
- Selecting the 'most appropriate' solution
- Synthesising a product
- Validating the product

... and possibly many stages of analytical learning, feedback and iteration.

Key to this technical realisation is the 'engineering judgement' stemming from underpinning knowledge, understanding and experience that is used in the analytical ‘selection’ process. These are the stages where immaturity, inexperience or lack of domain understanding can rapidly introduce significant technical debt.

Note that poorly engineered solutions may still yield a functional solution, but possibly lack all the desired attributes.

Whilst these address the technical issues, no engineering solution exists without a commercial imperative and an innumerable number of societal, environmental, political and economic constraints. Therefore we have to consider the impact of any potential solutions against:

- Market (users)
- Value (the benefit to the user, and what they are willing to pay for that benefit)
- Competitive solutions and Intellectual Property
- Development time (and cost)
- Product cost (Bill of materials)
- Product Lifetime
- Warranty/Service costs

The point is that this engineering regime is as applicable to software as it would be to a mechanical engineer.

Engineering the Real world

In my career (almost entirely in real-time embedded systems) in all my roles as Systems Architect, Designer and Engineer (Electronic

Control Systems centric), Software Architect, Designer and Engineer, that 'underpinning knowledge' has had to extend beyond pure software to the physically attached systems.

This has typically included a more than rudimentary understanding of multiple physical disciplines (notably electrical, magnetic, mechanical, hydraulic, pneumatic), Mathematics and Data Science, coupled with significant understanding of Electronics.

Computer Science and Computational Methods are my 'home turf'.

I find I regularly interact with the company experts in the various physical disiplines and have to mentor or coach Software Engineers in the realities of sensor physics or chemistry, electro-mechanical actuation mechanisms, or understanding basic physics of linear and rotational systems (e.g. inertia, momentum, force, torque, energy, power, work) as well as the applied maths and how they relate to 'time' over which the software engineer has some control.

For this 'immersion' over the last 40 years, I feel extremely grateful. It has continued to satisfy my thirst for knowledge and improved my 'mental models'. As a result, I now realise how little I really know!

However, my world is inherently 'unclean', frequently surviving on the most meagre data that is acceptable, at the minimum precision warranted to maximize robustness and minimise cost.

Where the Data Scientist lives with expensive lab-grade sensors, minimal noise and clean datasets, with identifiable data correlation, I am frequently:

- Looking to extract a trendline from sensor measurements where individual readings may be completely swamped by electrical noise, mechanical, electrical or magnetic hysteresis;
- Unable to use complex filters as the significant lag or computational complexity is unacceptable in spotting a fault, or delays action in providing a machine response;
- Making measurements where the data can vary over 5 or more orders of magnitude (the extreme being 14 orders!) and still need the last 2 bits of precision resolution;
- Faced with cascade failure affecting the data from simple faults yet recognition of root cause is a key goal, even if the sampled data arrives out of order;
- Using synchronous sample timing for measurement whose timing may be more critical than the actual value;
- Dealing with system decisions whilst elements of the system may be attempting to adapt and defend from failures or undesired effects before I am aware of it, masking the raw reality (and information) that might inform a higher-order adaptive model.

Confusingly also my sensor sets are often mis-named – a speed sensor doesn't typically measure speed (but captures trigger events, usually a measure of relative distance, and may also capture direction).. and even then the notion of instantaneous speed (time between 2 events), short term average (the time between a selected short run of events) or average speed (time for a number of events related to say a single shaft revolution)... are all 'notional' but may be relevant, and that their derivatives (e.g. acceleration) are fraught with mis-interpretation because they are discontinuous samples.

Is Software "Engineered"?

Not all software is 'bespoke'. In fact I suggest that for many software developers (even professional software developers) a significant amount of their work is 'borrowed' from other sources, if not outright (with due consideration for copyright) copies, but certainly in design approach, structure, computational method or idea.

If such 'research' is in support of gaining the appropriate knowledge and experience, that is good... but if it is 'uninformed re-use', then that has potential problems.

This same problem can apply to publicly available material and even professionally supplied or commercial software components.

This appears to be at the fundament of the difference between a 'software developer' and a 'software engineer'.

Supplier's Engineering and Warranty

From my work with high-integrity customers, we wouldn't 'trust' the engineering of any component (software, electrical, mechanical) being 'supplied' to include in our system implicitly (i.e. taken at face value).

Using such a critical component would require 'us' as a customer being convinced of the component engineering (analysis and process), the manufacturing process, control of quality and

performance and be subject to almost continuous (and quite intrusive) 'oversight' in that component's construction... and even then this would not preclude failures, albeit very infrequently, and typically from 'bizarre' (i.e. un-considered, usually extreme) contexts.

Even these failures would require significant investigation and feedback for corrective action (with the intention to not only understand other product vulnerability, but to eliminate as far as possible future failure cases).

If the IP was critical to the product we might look to own or make that component ourselves.

Few software component vendors today would accept that level of 'oversight' from a single customer, and it would be intolerable when trying to serve many different customers for anything other than 'commodity' product.

But underlying that principle of 'commodity' product is that it can be engineered once (typically in isolation of the specifics of the application), and the engineering context is acceptable to all future application uses. For a Safety Relevant component in Functional Safety this is often termed 'Safety Element Out-of-Context' (SEOoC).

Even In the mature world of mechanical engineering for example, mechanical engineers have different 'grades' of bolts with well-defined parameters that achieve most mainstream applications, but still require 'custom' fasteners for more extreme applications, or those with specific and unusual combinations of attributes.

Unless taken to very 'primitive' components (c.f. mechanical bolts) software parametric attributes for 'portable' (application non-specific) components would be very much more difficult to define and have significant inter-dependency on the interfacing systems (e.g. platform CPU, processor cache and policies, compiler, clock speed, bus widths, memory access times etc.).

Even common-place 'library' components are difficult to validate for all possible deployment circumstances.

Software (properly engineered) is NOT cheap

My perception is that software has for too long (i.e. at least since the general availability of microcontrollers) been sold as a 'cheap' solution for all sorts of engineering, without much concern of the (physical/mathematical) complexity of the problem. I too have historically, errantly, created the same trap with my (largely mechanically minded) masters.

Cheap, Fast, Flexible = over-sold

I once had to reflect back to some senior automotive C-level executives an understanding of what they were asking of a fuel control system, by showing the development of the non-linear controls they had 'required' ... (originally implemented via complex mechanical cams, springs, levers and dashpots, which quickly became unwieldy, heavy, uneconomic, unreliable and un-maintainable) ... and a proposed replacement with a very simple embedded solution (sensors, microcontroller, actuators).

The rotational and translational movements (in this case rotational speed, timing and phasing changes) were computationally quite low, with the equivalent (linear, 2nd order) 'maths' in software being relatively simple and easy to validate, but would allow them much greater manufacturing flexibility (ease of variation), calibration savings, warranty savings etc.

But that 'simple' solution immediately led to a runaway for the designers, with HUGE growth in control complexity (non-linear relationships, high functional dependency, modalities, security integration from the System Engineers. This assumption used the same argument as I had, that these additional 'simple' control laws, all could be implemented, at low cost, with some software.

At this point we had lost sight of the complexity being requested, because I had made the original transition look so easy (and so profitable!) as the design was easy to comprehend.

Unfortunately the reality is that complex design costs money, to conceive solutions and validate, no matter what implementation technology.

I tried interventions that related the complexity to their (mechanical) backgrounds, including the fact that in either mechanical or electrical control solutions it would have been, like Babbage's machine, both:

- hard to conceive
- untenable in size, weight, cost, friction

However 'software' had been sold as having no additional 'componentry' costs, having ultimate flexibility (relationship between input and output) and by definition it was all now cheap and simple.

Worse still, the robustness of an electronics and software solution with additional simple (cheap) sensors could now easily enhance the functional dependence to allow a much wider operational range, better performance or even operate in a more adverse environment.

The lack of a physical manifestation of software complexity (change to weight size or componentry) to mechanical engineers made this a wonder solution.

In my 'sales pitch' of selling the software solution, I had failed to establish that the (software) engineering was still as complex and costly as that required for a mechanical solution.

The implementation was the lowest part of the cost make-up.

Prototyping and Product Engineering

Systems engineering of novel designs (which frequently include electronic and software control systems) are necessary. Only simple, or re-applied systems, get to be right first time.

System engineering is also frequently necessary to 'explore' the sensitivity of the control of the real world with sensor numbers and practical (usually not ideal) placement, highly non-linear actuator relationships (such as the control mechanisms for driving an electro-magnetically initiated, hydraulic servo-controlled high-pressure actuator in fluid systems with asynchronous pulsed pressure and multiple, phase varying, dynamic, fast opening/closing 'taps').

During these prototyping stages of System engineering we knowingly omit some attributes of the design to focus on the aspects to be analysed as relevant to the system in 'discovery'.

To me, this is the analog of a mechanical scale model, or a space model, in an inappropriate material , e.g. plastics used for mechanical prototypes to judge key dimensions.

Unfortunately for software systems, those 'unreal attributes' are not so obvious and all too often a product becomes littered with the (albeit deliberate) naiveties of a prototype as the software becomes 're-used' as it does 'most of the job', and 'It's a good start point'(!).

Using production software development techniques on prototype code is, typically, prohibitively expensive. Generating a suitable Software Architecture for future use in product, clothed for prototype, is often infeasible as by definition the System design is not fully realized and stable (if it is, then the engineering has probably already been done, and the System is not novel).

The trick with prototypes is to 'learn' from them, yet NOT to consider them product engineering. It is also perfectly reasonable to 'learn' what software solutions perform well, or are most appropriate, and use that knowledge to feedback to the architecture and high level designers as key constraints.

Re-use at this level is 'informative' of design, NOT of implementation, which may have to consider many more detailed additional attributes.

Software Re-use - Myths

There are some well-established ground truths to Software Re-use and the (additional) cost of making software 're-usable' as a design goal (e.g. through Product Line management of intended variation to achieve certain combinatorial solutions).

However, the most often mis-guided view is that ad-hoc re-use (typically of a design or just code) is a simple and cost effective route to substantial saving, improvement in reliability, or any other goal, without first considering what the components were originally designed to do and the engineering evidence that supports them being viable in their new 'role'.

This is as true of software re-use from in-house components, open-source software or even commercial product that is sold as 'commodity' or 'configurable'.

It's also just as true when moving from Prototype to Product. The additional attributes are integral to the design and can't be simply 'added on', in the same way that Quality, Safety, or Security cannot be simply appended.

I wouldn't re-use even an 'elemental' mechanical bolt or machine screw in a new application based solely on its apparent functionality (thread pitch, diameter, shoulder etc...), so why would I do that in software?

Which Engineering attributes are valued?

In my professional software career across many industries, some highly regulated, some more 'cottage industry' whose output bore more resemblance to 'prototype' than product, there have been many 'sensitivities' to the priority of which aspects of engineering were valued.

In all cases there were always still more engineering attributes that never made the list (typically accepted as a commercial risk).

For instance, in Automotive today I see 'Functional Safety' (reaction to component failure) struggling to make the mark in appropriate System or Software design... and little penetration of 'Safety of the Intended

Function' (SOTIF) (action due to insufficient functionality, or foreseeable mis-use/mis-operation).

The latter SOTIF issue might also suggest heightened sensitivity when replacing 'human operation' with a 'rule-based' system that is not 'deducible' by a human.

I suggest that challenging examples will exist when considering the more statistical or probabilistic 'decision' outcomes of data analytic methods such as Machine Learning or Artificial Intelligence.

Bigger Engineering challenges to come

The existing lack of engagement on 'engineering' factors for software may prove to be 'minor cost excursions' compared with the dynamic cost 'feast' that is digital security in all its forms (privacy, confidentiality, integrity etc...).

For any embedded system that is expected to have longevity in the market, these engineering costs are significant, both at original product engineering, but significantly in the cost of maintenance.

Even assuming the end-user pays for the engineering of the update, the system cost (transmission time, memory requirements, transmission security, energy usage) may be a significant element of the original engineering design consideration.

For the assumed 'cheap' (to purchase and deploy), wireless connected, self-powered, IoT type systems , they will become a dominant cost that is only sustainable through sheer volume of product sales.

Product Cost vs Cost of Ownership

Traditional (since 1980s, largely isolated, single purpose) embedded systems software was for most cases assumed to be invariant over its lifetime. The software maintenance cost was therefore zero. This is reflected in the warranty and maintenance cost expectations.

With no lifetime maintenance, and precious little 'manufacturing' cost (programming an image is near zero cost), embedded system software costs were those of development costs, and for many industries these were recovered as part of the final product (seen/sold as electronic hardware) costs (the software load not differentiating the product value).

Driving the cost of such embedded software developments down was therefore key to 'productivity' and 'profitability', often precluding significant 'engineering'.

Functionality and Self Diagnosis

Historically this was possible, as the functionality (software task) assumed a static configuration of componentry (including any redundant capabilities for failures, where necessary) and an unchanging (mostly rule-based) requirement on behaviour.

Electronic component failures, like software errors, happen fast, with typically no discernible 'onset of failure' precursor. Condition based monitoring of these components is essentially useless, so an appropriate reaction to failure has to be clearly engineered.

Self diagnosis typically majored on failures of actuators and sensors (real-world interfaces, more likely to wear-out, corrode or be vulnerable to overloads or mis-use), rather than internal components. The emphasis of embedded system diagnosis is typically on providing a service engineer with sufficient fault information to identify, replace and validate any maintenance requirement.

Any 'variant' functionality was accommodated through simple configuration or calibration and was pre-validated, and accepted as correct for the operating environment.

Future Proofing

In today's climate, 'future-proofing' may explicitly require software updates for an otherwise pre-constrained system (electronic components, sensors, actuators).

This poses major (and costly) validation issues:

- the deployment context and system connectivity may be so diverse to have no guarantees that product validation (the read-across of validation evidence to a new context) can be argued as acceptable;
- the dynamics of the connected system may be an uncontrollable feature (e.g. due to wireless nature, route associativity, signal paths and strengths, router response times);
- the connected system interactions may be under continuous evolution from development, or adaptation due to algorithmic optimization (e.g. signal to noise ratio adaptations, signal 'signature' for security)

- other components of the larger system may also be replaced, without notice, with changes in functionality.

In this climate it is unlikely that we can design to accommodate all possible futures, and even If we could it would be uneconomic and impractical to validate for all possible combinations of deployment over the product lifetime.

This implies that every change in functionality (new feature, even on existing hardware) has associated with it an engineering cost in both design and validation of that feature which is not associated with the original product hardware.

Cost Recognition?

Does this mean that Software will at last be recognised as a 'cost', independent of the electronic hardware in the embedded market?

Further, some requirements, such as Security, are by their nature evolutionary, suggesting a periodic cost of software ownership to secure non-isolated systems from evolving threats, either directly, or by virtue of their connectivity.

Who Pays?

There appears to be a real blind spot amongst producers of these traditional embedded software systems to the difference between engineering development costs (historically the most significant costs to amortise into product sales for embedded systems) versus the maintenance costs (including any re-validation of changes) of 'connected and maintained' systems and especially those that are likely to go in that recurring cost category (e.g. 'Safety and Security').

We could of course build like Isambard Kingdom Brunel, in a horse and cart era, and over-engineer construction for the future heavy-weight needs, that amazingly seems engineered sufficiently to deal with vehicles 200 years into the future.

The reality however is that even in Brunel's time heavy horses and laden cart (total 10 tonnes say), versus modern articulated rig (max 44tonnes in UK) suggest only a factor of 4 to 5 change.

Although a brilliant engineer, even he met with 'the prohibitive costs' of engineering.

The other major differential in modern markets is the pace of technology adoption, which means a 100-year future-proofing in electronics and software would not be credible; even a 10-year, non-disruptive pattern of change would be exceedingly challenging to proof against.

Modern consumer electronics is often designed for a half-life of about 2 years for portable devices and 5 years for larger (electronically controlled) domestic white goods. The general consumer would possibly expect those goods to have a realistic service life of double that.

In the UK, vehicles are scrapped, on average, at about 14 years, with the average vehicle age (on the road) of 8 years.

The suggestion of a 10 year life for future-proofing thus seems a very reasonable expectation.

Automotive software example

There are already precedents that suggest that ownership of software will become significant.

Automotive testing and inspection is already suggesting that the annual 'pass' will require that "all available software 'patches'" be installed as a test condition. This suggestion already places an onus on the vehicle manufacturers to publish (at least to the testing authority) 'latest patches' within a defined deadline of learning of an issue.

It is unclear who becomes responsible for ensuring that the patches are a suitably compatible set, for the vehicle, or with each other; who has responsibility for installing them, and validating that they still provide all possible functionality, functional safety etc. or who pays for both of these issues.

All of this assumes that such 'updates' are able to be accommodated in the existing electronic platforms.

The Cost of Professionally Engineered Software

Software development, as an industry in all its forms and all its applications, is a HUGE remit, and as previously stated, whilst I have no problem with non-critical or low-impact failure systems (such as websites and some mobile Apps or IT systems whose loss or mis-operation is merely 'inconvenient') being written with less professionalism (even economically constrained), I take great exception to software that is used in devices that safeguard, protect and secure me, my family, my fellow citizens and the planet we live on being inadequately engineered.

The real problem is who is willing to pay for software to be 'engineered' properly?

Can we afford NOT to make that investment in critical applications?

Can we afford for such systems NOT to be developed by ‘engineers’ who can provide evidence to substantiate that the design and product validation shows it ‘fit for purpose’?

Even in the mechanical world, a large part of the population are still willing to accept unwarranted ‘re-manufactured’ replacements in almost every system, as a cost saving, without much thought for the original engineering (and validation) that went into the original part design and manufacture.

Even engineers succumb to this ‘economy’ when they are capable of comprehending the risk.

As Engineers we should at least consider whether a ‘look-alike’ fulfils all the same functional and non-functional criteria of the original part, and the implications of its failure to do so.

A good Software Development Process does not ensure good Engineering

Whilst having a well-defined (and hopefully mature, incorporating feedback and some elements of 'correction' if not 'optimisation') software development process helps ensure the 'quality' and ‘repeatability’ of the software development (production) process, it contributes little when considering the suitability or functional ability of the software solution to deliver the requirement, or solve the original problem.

Some of that validation of the appropriateness of the solution is down to good Systems Engineering, but even that doesn't ensure the Software Architecture and Design is appropriate, that the structure and behaviour of the software solution is appropriate, and can support the expected future flexibility of configuration, performance, or scale.

A highly supportive software development during system engineering, as discussed in earlier paragraphs, can learn substantially from prototyping and ‘inform’ the design of solution efficacy.

Prepared by

SOFTINTSYS

Software and Systems Engineering Consultants

LANDLINE: +44 (0) 1283 575609

MOBILE: +44 (0) 7913 205457

EMAIL: info@sofintsys.com

WEBSITE: www.sofintsys.com