Architecture: Mitigating Software Degradation, Wear-out and Obsolescence

Degradation, Wear-out and obsolescence; These are not terms naturally associated with Software. The traditional thinking is that software is “fit and forget” in as much as it remains consistent in operation (i.e. invariant) throughout its life. But if the software product’s life is intended to be, or becomes, prolonged, or needs to adapt to changes in its own environment (as a result of other system changes, developments, or obsolescence) then this investment may not be “once, for life”.

The anatomy of a Software Error

Software flaws (whether design or implementation) are inevitable because of the human process of production. Whether these errors manifest themselves as faults depends on the application being exercised through the faulty regime, with the appropriate conditions. If the precise conditions are met, then the fault will manifest – a very deterministic response. The unpredictability of the fault becoming manifest largely being a property of meeting the trigger conditions, which usually are multi-variable dependent and the result of significantly complex sequences of events.

Contractual repairs

For most ‘dependable’ systems, correction of such flaws is a system (service) contractual expectation. Today however, few systems have to warrant software operation, and of those, few are expected to pick up the economic “tab” for the catastrophic potential that its mis-operation can cause. In most ‘shrink-wrapped’ software the warranties explicitly exclude this liability and the typical fault mitigation is “upgrade to a later version”, often through a complete repurchase (depending on license model).

Corrective deliveries

Well- and web-connected devices may have contracts that get ‘service delivery updates’ that ‘maintain’ the software as part of the licence agreement, and the user gets minimal choice on what ‘corrections’ he receives or with what else they are packaged, often only able to choose the timing of their introduction (and possibly a capability to ‘back-out’ of the correction to a previous state).

Wear-out and the effect of software “Maintenance”

Unlike mechanical maintenance, where ‘repairs’ can often re-instate the reliability period, “Fixing Bugs” in software is just as likely to introduce new errors – especially for a complex flaw. Often software vendors like to combine (or even hide) bug fixes through introducing “New Features”, themselves fraught with more potential for introducing errors.

Architectural impact of Maintenance: Degradation

A good original architectural strategy allows the mitigation of those themes for which the architecture was designed to accommodate. Good architectures are easy to change, but applying ‘good corrections’ which live the ethos of the original architecture are rarely applied... as there is an attendant cost to this “purity of purpose” and often a significant additional intellectual effort.

Maintenance of a software product as a result of corrective actions can, all too easily, cause pollution of the architecture’s intended goals (this process is often referred to with a mechanical analogy of ‘rusting’ or ‘corrosion’ due to the way it attacks the surface initially with superficial effect, but eventually undermines structural integrity). Rarely are the change agents aware of the original design intent, or have access to the original design information or team. Frequently “software maintenance” is deemed a ‘lower skill’ task, becomes an immediate ‘legacy’ issue and is scheduled for offshore maintenance where access to, and understanding of, original design rationale (intent) is even more remote.

Maintenance of Software and Lifetime

For short-lived and disposable software products architectural maintenance is typically about economics of design and opportunities for re-use, rather than the cost of support.

In long-life products architectural maintenance can become the dominant problem. Mitigating the cost of obsolescence of the electronic hardware, notably the processing fabric, are significant objectives of the architecture.

For safety systems, architectural support for preservation of the integrity of operation and an ability to ‘compartmentalise’ the impact of change to reduce change propagation, enable the re-use of previous verification evidence and maintain confidence in the solution, are all high on the agenda.

Architectural maintenance for large or complex systems, enabling the solution to be partitioned for concurrent, potentially remotely dispersed, developments and/or coherent changes, is a significant driver. Once again the impact of propagation of change is a major factor.

Future “Predictable” Software Systems

As our future marches ever onward towards larger systems-of-systems, of ever increasing scale and connectivity, integrated systems of mixed security and safety integrity are inevitable. Predictable operation of these systems means that high-integrity component systems must ensure primary goals, utilising defensive means and ‘bounded’ co-operation with the larger system. In such systems, degraded operation of the larger system, whilst protecting the predictable (safe) operation of the assured system, is usually the preferable system response.

The advantage of such systems-of-systems will usually be realised from greater overall performance (in terms of features offered to the user, rather than execution or resource usage), but only where such systems are well-behaved, even in the event of failure, mis-operation or malicious intent.

Platform migration

Architectural transparency is already key to migration of applications across different platforms, usually concentrating on changes of microprocessor core in an otherwise like-for-like scenario. In future architectural transparency needs to mitigate changes in platforms that cause migration from single to multiple CPUs for application environments, or even to enable transparent operation across networks of platforms of potentially multiple heterogeneous CPU cores, in dispersed applications. These states are likely to be the signature of evolutions of large scale systems where ongoing renewal of components of the system are inevitable.

Wear-out and Obsolescence by Architecture

At some stage our current thinking, and architectural response, will not deliver sufficient future flexibility to accommodate our changing world. The speed of change for software (and electronic) systems continues to accelerate, driven by consumer demand, the green agenda, control of, and access to, information and the complexity born of the demands for high precision, discontinuous controls and instant response.

For software systems with long (decades) of expected life I suggest that moment is already here.