...Cash

For Finance / Business / Commercial / Procurement audiences

Introduction

There is a belief that, for Software, the engineering costs of design and manufacturing are largely up front... duplication/distribution are the only remaining cost. This is, of course, nonsense. It appears to be based on an assumption that software is perfect and that its behaviour post-delivery is always fault-free (i.e. that there are no residual errors, or that those errors will never manifest themselves as faults)!

Software costs come in 2 parts, the original engineering (including manufacturing for software)... and then the cost of maintenance through-life.

Through-Life Costs from the software itself...

Software Development is a Human endeavour... and humans make errors... the process of development is imperfect... therefore all software contains errors.

The number of residual errors (i.e. still present post development and its verification) has a number of factors including product architecture, process rigour, competency of the engineers, complexity of the product, effort expended.

In software, design, manufacture and commissioning are all part of a single process... the remaining processes are largely distribution, installation and operation which do not impart any variability.

... and software's initial Development Costs

The prime drivers of Software Engineering Costs during development are Size and Complexity, Delivery deadline (as this drives the resource non-linearly), Competence and Process (usually differentiated by rigour). The general perception is that the latter is a response to either the regulatory environment or the market place (usually an 'industry best practice' argument).

In general, putting more rigour into the development will reduce the incidence of through-life costs due to the software itself.

Through-Life Costs for Software also come from "elsewhere"

Software is 'easy to change' (relatively, when compared to say mechanical change) and it is this property that makes software's flexibility a significant advantage (and curse!).

That advantage has ensured that in embedded products, any change that is traditionally 'difficult' (i.e. costly) because of the rigidity of the design process (mechanical, electronic), may offer an advantage if it can be solved in software or at least a programmable solution (which inevitably is analogous to software).

Extrapolating that thought further, why would you not design as much of the functionality as possible in software/programmable solutions in the first place, to give you maximum flexibility?

Lightweight processes are characterised by agility, responsiveness and a short shelf-life where beating the competition is determined by 'time-to-market' (where 'brand allegiance' competes directly with 'personal value' determined through 'feature set and price point' to the individual buyer).

Cash - Flow

Obviously high-rigour processes spend a lot more up-front money and are much trickier in terms of DEVELOPMENT cash-flow as it may be a long time before the product reaches the market and starts to earn revenue... which generally associates high-rigour processes with markets where:

- The consequence of product failure is >> Cost of product engineering
- Warranty of behaviour has significant financial consequences

High-volume product that has a relatively moderate safety element (e.g. an automotive braking system or stability control) tend to be adopters because of the 'volume-risk product' consequence...

... whereas low-volume, high-safety equipments adopt because the consequences can be damaging in a single instance.

The warranty of behaviour of software often carries with it consequential costs of associated damage (consequential damage, injury claim, repair or re-instatement of significant capital equipments, loss of service revenues) as well as reputational damage. This is often far more significant than the original development cost!

Low-rigour markets are often associated with shrink-wrapped software applications, where no warranty is expressed or implied (buyer beware)... and if it doesn't work, purchase the upgrade (or a competitors application!)... and the consequential costs of change (e.g. re-investing/switching operating procedures etc.) remain yours as a customer!

Realities

As product complexity grows, businesses will look for more opportunities to provide solutions in software, at original design, or for protecting the cost of correction "in-service", making enhancement potential easier and cheaper to deploy.

Public and customer expectations are for increasing reliability, security and safety from all software enabled products, but ever more so from programmable equipments, because of the capability being marketed for their consumer devices they are attached to!

Cost of ownership of software post-development is no longer just about the failings of the original software design or implementation, but increasingly about its ability to 'soak-up' problems from elsewhere in the equipment, product, or even elsewhere in the system... either temporarily in fault accommodation, or long-term, by controlled mitigation or management of product design flaws, inefficiencies or unsafe areas of operation.

Managing software development is difficult, and has a chequered history of delivery, quality and cost in practically all industries... but it is here to stay and in substantial growth in all of our businesses.

We need to get better at understanding the risk and cost of ownership across whole-life, using system and software product architecture as an opportunity to minimise impact of change in a fielded product, whilst maximising the opportunity for product or system enhancement for future revenue, or to maximise product longevity.

That means not only estimating, managing, and delivering to match those aspirations during the initial development, but taking a more systematic view of what software might reasonably be expected to accommodate through life.

This same premise suggests that we should be VERY conservative with initial processor or micro-controller occupations for initial fielded product. The cost differential between a processor that can 'just manage' to do the job, an done that we utilise only 10% of at first fielded product is insignificant to the Bill of Materials cost, but may be significant in its ability to mitigate expensive on-costs, let alone the amortisation of the engineering costs to accommodate such changes. Spending engineering effort to 'squeeze in' software functionality is pure folly at almost any stage of a software enabled product's life!

Components of Total Cost of Ownership of Software in an embedded product

Market Research – Cost of identifying the opportunity and its value, likelihood of capture etc

Technology development – USP for the functionality of the software product?

Sales and Marketing – Developing customers, Promotion

Acquisition Costs – Cost of original Purchase and/or Development (including Test)

Application Costs – Costs of configuring, building, testing and deploying to a particular customer.

Manufacturing costs – Costs of replicating, packaging and distributing software, along with management of same.

Operation Costs – Investigation of Safety or Security failures, mis-operation,

Warranty Costs – Direct Cost of correcting defects in the software, or Indirect Costs of accommodating design defects in the system by changes in software.

Liability Costs – Breach of warranty, consequential damage, disruption etc.

Licencing Costs – Rights to use 3rd party software. Tracking use of own software if licenced.

Maintenance Costs – Cost of managing obsolescence of either the microcontroller (direct consequential changes to software) or accommodating obsolescence in system components (indirect consequential changes in software)

Upgrade Costs – Costs of extending the capability to maintain place in market.

