Software Engineering Related to Systems Requirements

Dr. Richard John Stevens,
Information Systems Division (ISD)
ESRIN, European Space Agency, ESRIN,
via Galileo Galilei
Frascati 00044,
Italy
Tel: (+39) 694180280
Fax: (+39) 6 941 80 361

Abstract: Current software standards (rightly) take a reductionist view of the methodology needed for software development. But for developments of large "software systems", the needs for commonality, or developments of software within the systems lifecycle, current standards may not be enough. Some help towards these problems may be available by abstracting the systems engineering process. ISD has been developing a structured approach to handling systems requirements, and this has revealed many of the problems and incompatibilities between the two disciplines, but also provided some pointers to help complex software developments.

1: A Comparison between Software and Systems

Software engineering and systems engineering share many similarities, but there are also great differences. Software engineering has learnt to handle complex detail, and is in many ways more advanced than systems engineering.

However, systems engineering also has many powerful techniques which have not been academically analysed. The review process, contractual approach, message passing, information-hiding, object-orientation, handling of risk, design qualification, are examples of powerful systems-level paradigms. Some have only recently been 'discovered' by software engineering, and even graced with new names.

When software is developed as an integral part of systems, the two disciplines may have conflicting constraints on each other. Software has its own, rather rigorous lifecycle. Under a system development that lifecycle may be put under severe pressure, by for example, delays or changes in the system. In practice most software developments will now contain systems elements (such as operations) or face complex management issues (such as common developments). Even an information system will be tied to the project schedule. These complications are beyond the scope of simple software methodology standards.

Systems engineering has developed a wealth of techniques over many decades, while structured methods in software can be adapted and re-applied at the systems level. Each can learn from the other. Software and physical systems contain many similar aspects, as well as surprising differences. As software becomes more complex it will need to adopt more of a systems approach. Some future trends in software standards may be deduced by looking at system engineering life cycle. The differences between the two engineering methods are illustrated by current problems in developing software-intensive systems, i.e. physical systems to which software makes an important contribution. Spacecraft are an example of hardware and many different software elements. The software engineering standards in current use are insufficient to describe the production of such "software systems" in a mixed software-systems environment.

1.1 A Requirements Model

In support of ESA Programmes, ISD has developed a system requirements model [1], which can be compared and contrasted with the Agency software engineering standards (PSS-05). The requirements model is the beginning of a systems methodology. This model is not an invention but a structured compilation of knowledge extracted from dozens of systems engineers, from inside and outside of ESA. The model covers the acquisition process, rather than the manufacture, i.e. is designed for ESA use, not its contractors. Experience in the model has been gained by applying to various ESA projects, both for software, systems, and for software-intensive systems. The requirements method is "upwards-compatible" with the Agency's PSS-05 software engineering standards [2].

This model has been implemented as a data base, with application programs to support various project activities. Requirements can be analysed, presented, and cross-referenced, and specialists such as software or
structural engineers can extract selected elements from a coherent information source. The model consists of inter-related tree structures, which reflect the project phases, logic, and traceability approach. A structured approach to systems development is built into the model.

The basic principles behind the approach are therefore:

- Partitioning requirements into well-defined sets.
- Assigning single-point responsibility for writing each set.
- Splitting requirements sets hierarchically.
- Separating functionality from design.
- Cross-referencing all other requirements sets to functionality.
- Controlling the quality of individual requirements.
- Partitioning and reuniting requirements sets uniformly.
- Driving the project through requirements milestones.
- Defining a requirements lifecycle.

ESA is not an organisation which builds satellites. Instead it requires them from the contractors who actually build them, and then manages those requirements until they are satisfied. Requirements have many advantages as a control mechanism, providing they are well structured. They are complete, fairly non-technical, small in size, available early and continuously throughout the project life, allow delegation to contractors without interference in design responsibility. For these reasons the methodology of handling requirements is a key task for an organisation like ESA.

1.2 The Software Methodology

The software engineering methodology is much better defined. The basic phases of the European Space Agency software engineering standards (PSS-05) are:

User Requirements - the definition of the results expected.
Software Requirements - the abstract solution to the problem.
Architectural - the top-level description of the specific solution.
Detailed Design - the detailed definition of the specific solution.
Acceptance - the series of checks that the software performs as expected.
Transfer - the final testing and preparation for operations.
Operations.

PSS-05 is an "adaptive waterfall" model, which allows controlled feedback between phases and in the 1991 version also encourages the tailoring of the standards to manage specific problems in the lifecycle. Another major point is that PSS-05 is a minimum standard -it defines the minimum that is essential and assumes the software engineer possesses enough common sense to adapt the standard to particular needs. The standard is also for the development of a single software entity, not for multiple linked pieces of software. These assumptions keep the standard short and readable. The standard is also at a

---

**Figure 1: The Requirements Model**
good level of granularity - defining a project as a few phases with very clear outputs. This makes it sufficiently understandable to be modified to the needs of an individual project. Once it has been understood, it acts as a starting point for altering it - just like a good recipe.

It should be immediately clear that there is not a one-to-one relationship between the systems and the software lifecycle. In a satellite project, for example, software will be needed from the earliest development phases, for tasks such as supporting the project reviews, requirements handling, CAD/CAM, and helping communication between the project participants. The majority of this software will be bought on the commercial market, and not developed specifically for the project. No other method is feasible for software needed at the beginning of the project - unless there is a support department which has generated suitable software for previous projects. At the later development stages, software will be required for testing the spacecraft, launching it and controlling it. Software will control the spacecraft, its subsystems and the experiments carried on board. On the ground, scientists will use software to monitor their equipment from their laboratories, and to process the results after the satellite has finished its mission. The lack of linkage between the two lifecycles must be obvious.

2: The Differences between Software and Systems

Conceptually there is little difference between the software lifecycle and the system lifecycle at this high level of abstraction. However, when the software development is compared in detail to systems development, the differences begin to emerge. Some are differences of emphasis, but the sheer number and scope of them should give a software engineer pause for thought before thinking of systems as trivial extensions of software. They should also make the systems engineer think carefully about how software needs to be managed within a systems environment.

A list of the differences between software and systems is now given.
1: Systems engineering handles disparate disciplines each with their own complex technical languages. It must communicate with them all, preferably in as similar a way as possible, using a general control board. Software controls one much more homogeneous discipline

2: Systems engineering is primarily a management activity; software engineering is primarily an implementation activity. The culture of software is to receive requirements from above, not to capture and manage them.

3: Systems engineering handles multiple semi-independent sub-systems. The objective is to delegate as much responsibility as possible, while still coordinating them at the requirements level.

4: Constraints, as engineering requirements, are much more important for hardware.

5: Software interfaces have more penetrative power than hardware interfaces.

6: Software can be changed at late stage during a project. On the surface, software is apparently much easier to change, but change is also very dangerous and potentially expensive.

7: Software has no manufacturing stage - it is all a design process, and the risk is therefore all design risk. The ease of copying alters our practice towards software. Multiple historical versions can be held. Testing can be performed in parallel. Multiple copies of software can be rapidly destroyed by accident or by deliberate sabotage.

8: Software progress is difficult to measure because the working system may not appear to be late, and is difficult to visualise, unlike say, architecture or mechanics. Progress during the development process is difficult to make visible.

9: The relationship between the functionality and physical structure is closer for software than hardware.

10: More risks tend to be taken during the development of software because it is perceived to be easier to correct errors.

11: The working environment of software is an abstract, uniform machine. For example, temperature and humidity are irrelevant to the software, but not the hardware. Software can be generated to be somewhat independent of its "external" environment (e.g. in a portable language).

12: Software cannot be damaged by testing. Therefore operational software can be stressed repetitively above nominal test conditions to see where it fails. In hardware, we must not overstress components which will be used operationally.

13: Software is inherently serial, whereas hardware is normally parallel. In a design, the same software is reused. In comparison, hardware has to be replicated - a car has four wheels, whereas the equivalent software would re-use the same wheel four times.

14: Software can be upgraded without the need for physical contact (e.g. spacecraft software, for delivery of updates, for transmission across borders)

15: Every software development is a new development (if not why not make a copy of the old?). Coupled with the difficulty of measuring progress, this means that the costs cannot be estimated accurately until later in the development. It should be easier to cost hardware, such as a building, because of the experience there is in building similar things.

16: Software can be replicated from the final version, but not maintained, but with hardware the opposite is true.
17: Maintenance to software creates a new design and a product, whereas for hardware it is aimed at returning to the original design and product. All changes to software create a new product requiring re-qualification. Only one master copy of the software need be maintained. Software does not decay and can be returned to its original state easily.

18: Software is normally maintained at the same configuration level as it is developed. Hardware is maintained in the units used for repair.

19: Software does not cover the complete life cycle of a system; the user requirements stage, which is a difficult and demanding area, is relatively neglected in software.

20: Software engineering has a history of only a few decades. Few systems managers have a successful software background and consequently there is little understanding of software at the systems level.

21: Software engineering has powerful paradigms such as reviews and contracts, which work well, but have not been taught academically, or defined intellectually.

22: External interfaces (and the constraints they impose) are relatively more important and more variable for physical systems.

23: Software is currently much less re-usable than hardware. For example, hydraulics systems and jet engines are used on different kinds of airplanes, but that is currently impossible for avionics software.

24: The working life of software is normally months before substantial change, while hardware such as a bridge or tunnel will be expected to last several decades.

2.1 The Differences in Detail
A few of the most important differences are examined in more detail to see their impact, starting with the fact that a systems is a complete project from beginning to end, whereas software may only be part of that. This gives systems engineers a wide understanding of the complete life cycle starting from customer involvement. Software engineers do not normally have the same range and scope, because they enter the project at a later phase. Systems engineering is not about implementation, but technical management - creating the requirements, managing the system development, partitioning the system, ensuring the interfaces, and sharing the risk.

2.2 Theory of Software and Systems
The theory of software engineering is now more advanced than that of system engineering. Software is complex and intangible, and therefore a disciplined approach has been found to be essential. But because all the process is performed on the computer, the process is inherently measurable. Moreover, the software lifecycle is at least somewhat constrained compared to the system lifecycle.

The best software developments are performed in a disciplined manner, but in practice many software developments are poorly controlled. Surveys suggest that a large majority of software contractors operate with an "ad-hoc or, possibly chaotic" process for managing software (Software Engineering Institute, Carnegie Mellon University).

Of course, standards on their own will never guarantee a good development. But without some coherent approach, failure is almost certain on any complex development. Only 2% of software analysed during a survey was usable as delivered [3]. Half of the software was never ever used - surely a good indication of requirements failures. The software industry is plagued by late deliveries of unusable software, and probably the major reason is the amount of novelty in the product.

2.3 Cultural Attitudes
Another different aspect of culture is age and attitude of staff. Especially on large projects, the management and systems staff tend to be at least middle-aged, experienced and slightly cynical. The overall attitude is against taking risks - large project cannot afford technological risks, and most engineers have seen failures caused by taking unnecessary risks. On pure software projects, most of the staff are young and relatively inexperienced. They are optimistic, prone to take chances on the development, and want to use the latest technology, even if unproved. Any books, software, or techniques more than five years old are regarded as ancient history. Both attitudes are correct - for the typical separate tasks of systems and software. When they are mixed together, problems arise. The systems engineers see the software staff as flighty, young and inexperienced; they in turn are viewed as reactionary, old-fashioned, and "low-technology". Naturally the systems culture must prevail in a system, but also the systems engineering culture must adapt to the new possibilities of software.

2.4 Flexibility of Software
Software is very flexible, and this a strength and a weakness. Performance in software can increase dramatically through the use of a new development tool, or a faster computer. Because of its inherent flexibility and the high rewards for risks, some software engineers are prone to take risks. If something goes wrong it can be "fixed", preferably on the fly. This attitude is dying in professional structured developments, and has never been acceptable in physical systems. Hardware is not so easy to change or modify. Complex system suffer from interfaces, and complex interactions. If there are ten elements and everyone takes a 10% risk of failure, then the project is almost certain to fail. Moreover, systems performance does not increase dramatically every year. This year's best design will not be significantly better than last year's, when we are dealing with ships, houses, aircraft, or bridges.
2.5 Attitudes to Risk
The rewards for risk are much lower in large systems. If the aircraft is delivered late, or the bridge falls down, the project manager is clearly responsible. The consequences are that, given a choice, the sensible project manager will take very few technical risks. There is already enough unavoidable danger in a large project - no one needs any more.

2.6 Different Types of Complexity
In a certain way, software is perhaps the most complex human product. A program may have several million lines of code, performing $10^{11}$ operations per day - millions of components used in combinations which have never been tested, and which can never be tested before use. While the complexity of software is in the number of similar, inter-related items that are managed, that of systems engineering is in quite another. Systems can fail from an enormous variety of potential problems - temperature, degradation, manufacturing tolerance, wearing-out, human intervention (or lack of it), humidity, theft, lack of spares, or the inability to change. The complexity of a system is derived from the range, and unpredictability of the problems it must handle. The fact that the dangers to systems exhibit more variety makes systems engineers more careful about taking risks.

2.7 Summary of the Differences
At the systems level, the life cycles of software and systems are similar, but there are fundamental differences between the two disciplines which are certain to lead to problems in mixed developments. The development of software becomes a very different process when it is exposed to the constraints of systems development. Similarly, systems engineers must understand the impact of software at the systems level, to adapt efficiently to the opportunities that software brings. Software experience must be available at the systems level. These differences are managerial rather than technical and appear as constraints which affect the way each discipline must proceed. The net consequence will be that relatively less software will be used within SISs, its expense will be visible from the start, but it will be built better.

Despite this list of differences, it seems clear that software management is becoming more like systems engineering. The old assumption of a "software system" which could be considered in abstract is very convenient. But in practice, software will not work for more complex software development. Such developments are necessarily more integrated into systems and into the real world of people, hardware, and external interfaces. Software projects are becoming larger, more disparate, and with more complex interfaces to non-software elements. The "software project" remains an extremely useful allegory for simpler projects, because it helps develop a single piece of software. But now the software manager will need more help than this to manage a complicated system-linked development.

3: Software-Intensive Systems
Software-intensive systems (SISs) are defined as projects where a significant part of the resources, overall system performance, or the development risks are due to software. The problems of developing software-intensive systems are well documented. Some of the difficulties can be illustrated by analysing where software appears during the systems development lifecycle.

No-one requires software per se - it is part of a solution to a problem. The operations requirements should therefore contain no reference to software. At the functional stage, it is not known whether to perform a function in hardware or software or indeed by people. This is a decision to be taken in the architectural or implementation phase. Therefore, in the simplest case, a single entity of software should appear at the moment that it is decided that a function is to be performed by software. At the same moment the equivalent hardware and the interfaces between the hardware are decided at least functionally.

In practice, on large systems, other complexities are caused by the needs to:
- a: Prepare a development system for the software.
- b: Re-use common elements of software.
- c: Use software during the definition and development phases.
- d: Prepare early versions of software
- e: Prepare advanced software required before the formal documentation is ready. [4]

4: The Baseline Lifecycle of Software within Systems
The development of SISs is now considered, starting from the simplest case and gradually building up in complexity. The simplest SIS is a case of a single entity of embedded software in a target system, then multiple entities, followed by multiple elements with commonality. After that the problems of software needed for the development system are analysed. The objective is to see the extra requirements that the complexity introduced within the SIS.

Software is required throughout the system life, forming multiple small "subsystems" which are needed at every stage from Day 0 to the end of the project. Some software will be bought commercially, and some provided by specialised support departments. Consequently, there is no generalised relationship between the life of every single software product and the overall system lifecycle. Nevertheless, for some parts of a system, software and
hardware usually have to be developed coherently. The most obvious example is hardware with embedded software. The two have to work together in the final system and so have to be developed in tandem, sharing the development risk sensibly. It would be pointless having one ready long before the other, or having excellent software which could not run on the hardware. Embedded software and hardware is therefore used as a starting point for exploring SISs.

Such embedded software will not be visible during the user or functional requirements stage. Only when the functional requirements start to be classified will it become clear that software is to perform some particular tasks. Of course, the approximate boundaries between software and hardware may be estimated beforehand. This allows a rough estimate of the scope and early preparation to be made. But the task of partitioning the two happens between the functional and design stages. Functionality is an abstract description of the system, which should specify "how" not "what". A function might be implemented by software, hardware, people, equipment or any mixture. Software is part of the solution not the needs. The interfaces to the hardware are also decided at the same stage.

4.1 A Single Software Entity
The simplest case of software in a system consists of one piece of software embedded in system hardware. This software emerges between the functional requirements and the design stage. Software should not appear as a user requirement or in the functional definition of the system. The allocation of functionality to either software or hardware involves knowledge of both hardware and software. The best choice might be software, hardware or people - until we see the functionality and its required constraints, the best solution cannot be decided. Figure 6. TBD shows the logic. Any number of factors may affect the choice of hardware or software - cost, risk, schedule, weight, flexibility etc.

This analysis is concerned only with the extra problems induced by a software in a development - the problems specific to systems or software are ignored.

The consequences of having even a single piece of software are:
1: The system architect must have the experience and knowledge to optimise the partition of functionality into software and hardware. The system engineer shall partition constraints such as risk and availability so as to meet the system constraints, and also to be realisable efficiently.

2: The software developer must understand and accept the constraints partitioned with the software task. These will include costs, schedule and risk. Constraints which preclude a good solution in software have to be detected and re-negotiated. There must be coherence between the development procedures for the system and the software. The system change control or quality assurance procedures may be different and not work well for software, and either have to be changed or rejected.

3: The system engineer must understand and accept the constraints that software is imposing on the system. For example, according to PSS-05, providing the requirements are stable, the software should be costable at this stage to +30%. This may not be sufficient for the system engineer, because hardware can be costed more accurately at this stage. The system engineer must accept that development of software within systems constraints is non-optimum and cannot be as efficient as without those constraints.

4: During the functional analysis, the hardware and the software developers shall define and accept their functional interface and, work together to define the physical interface.

5: The software manager shall ensure that no predefined hardware precludes software from meeting its requirements, or taking more than a reasonable share of the development risk.

6: The software manager shall ensure adequate margin to cope with the required variations in the system development. Once the development is under way, the software is at the mercy of systems changes. If the project is delayed or makes a change with a big impact on software, then the software has to cope. The software manager shall clarify and formalise what is expected from software in terms of flexibility, late changes due to hardware problems, and potential delays in the system.

7: At the design stage, the software manager must show that the software meets is part of the system constraints.

8: When a change in the system necessitates software change, the software manager must detect this, determine the impact on software, and either negotiate the extra resources necessary or reject the change.

4.2 Multiple Software Entities
The next complication is to assume an SIS, i.e. instead of just a single entity, the software is beginning to become more important. Multiple, but independent, pieces of software are required for the system.

The major differences as far as management is concerned are:
1: The systems architect must have enough knowledge to ensure that the software elements are really independent.

2: There should be an understanding of software at the user requirements stage, to ensure that requirements are feasible, and to ensure that opportunities available to the users from software capabilities are presented to the users.

3: The systems architect should have enough knowledge to guide the functional breakdown to existing, common, commercial, or easily realisable elements of software.
4: Any preparatory work for the software development (e.g. training, tool evaluation, staff recruitment) must have been completed by the beginning of the architectural stage.

4.3 Common Development Requirements
The next extension covers multiple software elements which exhibit some need for a common development approach. For example, if an operations service has to manage many different projects, it would be pointless handling them all in different ways. Combining the software needs into single acquisition can save money and adrenalin. This requires a commonality analysis at the user or functional systems stage. (Commercial software or software which is re-used from previous projects is ignored for the moment). The simplest case of this is for the target system. As with any design decision the basic idea must be to maximise efficiency, but commonality is even more important because software can be replicated at essentially zero cost. Software performing similar tasks may well be spread across different functions, and these will need to be collected together for efficient production. The process is familiar to software engineers who will re-use the same software to meet different functions, but in this case there must be formal "external" agreements with the managers of the subsystems who will use the software.

4.4 Summary of Software-Intensive Systems
Other complexities will be caused by the need to have advanced software before a formal lifecycle can be defined, or early versions of software to meet hardware schedules. As with almost all the other aspects, the analysis demonstrates that extra work at the planning stage is required to save effort later in the project. For example:

1: There should be sufficient time for software to be grouped functionally and a "software system" developed.

2: There should be sufficient time for a combined development system to be put into place to actually produce the software.

3: The software manager must negotiate with the managers of the subsystems to get an agreed commonality process for the development of shared software.

5: Summary
Most of the problems of software development in relation to systems are clearly managerial rather than technical. They relate to conflicts between the software and system lifecycles. Almost all of them can be solved only by a non-optimal software development, costing additional resources because

A major requirement for developing software in a system is a coherent system lifecycle. This is currently not available to the same quality as for software engineering. A structured approach is very difficult in current systems developments. The current theory of software engineering (even if not the practice) is ahead of software. Systems engineering can learn from this abstraction approach, adapted to systems needs. However, there are fundamental differences which make systems engineering different from the current view of systems engineering.

Systems engineering is not about implementation, but technical management - creating the requirements, managing the system development, partitioning the system, ensuring the interfaces, and sharing the risk. The same qualities are increasingly needed for managing software engineering projects - we can view this as meaning the software managers are becoming systems engineers, or that software is becoming more like systems. The need for re-use, for complex and more variable "software systems", use of commercial software, increasing complexity, the importance of constraints such as reliability, failure-tolerance, and robustness mean that software is becoming steadily more like systems engineering.

Successful development of software within a systems environment demands that software be allocated the correct weight at systems level, especially during the initial planning stages. The systems engineering team must ensure that the interfaces between software and the rest of the system are defined and implemented coherently, e.g. by preventing the hardware being defined too early, and blocking system changes before realising their impact on software.

References