Software Requirements Are Systems Requirements

Anthony Hall
Praxis Critical Systems

1 Abstract

Requirements engineering principles and practice are equally applicable to systems engineering and software engineering. There is, however, a belief among systems engineers and software engineers that the others' discipline is irrelevant or even contradictory. This belief is harmful. Each discipline can learn from the other when doing requirements engineering.

2 Position

2.1 The Problem

One of the more depressing phenomena in the requirements engineering field is the misunderstanding and outright hostility between systems engineers and software engineers. It is particularly ironic that this war rages in the arena of requirements, since this is exactly the area where the two fields should overlap most. Clearly the methods of software design are different from the methods of, say, mechanical system design; but this does not mean that we should express the requirements for a software clock differently from the requirements for a mechanical one.

If this hostility had no effect other than a certain professional protectionism it might not matter. In fact, however, it leads to serious misunderstandings between the two communities, and a refusal of one camp to learn the lessons of the other. Let me illustrate this by the fate of two excellent books – Jackson's "Software Requirements and Specifications" and the Robertsons' "Mastering the Requirements Process". The first of these is almost ignored by systems engineers, because it contains the word "software" in its title. However, there is much that systems engineers could learn from this book, and indeed it is my thesis that Jackson's key ideas unify requirements engineering in the two fields. Worse still is the fate of the Robertson's book: in spite of not mentioning the dreaded word "software" in the title, the book draws on some methods that have been used in the software community. As a result at least one self-styled systems engineer has condemned the book for using "softwary" notations such as context diagrams.

2.2 Systems and Software Requirements Are the Same

It is my belief that the essence of the requirements process is the same for *any* system, from a £6Bn railway upgrade to a few thousand lines of code in an embedded microprocessor. Whether the system is implemented in software or not, the nature of its requirements, their relationship to the real world, and the process of eliciting, analysing and validating them remains the same.

My evidence for this belief is twofold:

1 There is nothing in the theory underlying requirements engineering to differentiate between requirements for software and any other sort of requirements.

2 Our practical process for requirements engineering has been applied, with equal success, to a range of projects from pure software development to massive systems engineering efforts.

My first claim depends on there being such a thing as a theory of requirements engineering. I believe that Michael Jackson, in his publications from 1995 on, has indeed given us such a theory. Jackson himself makes modest claims for this theory, and in particular only applies it to software, both in the title of one of his books and in the examples and techniques he talks about. However, I would disagree with him if he were to claim that his ideas could not be applied in a wider context.

The essence of Jackson's theory – his $e=mc^2$ – is the observation that in order for a machine whose specification is S to satisfy requirements R when placed in a world described by domain properties D, the sequent $D, S \vdash R$ must hold. Now the requirements R are, by their very nature, independent of the realisation of the machine: they are effects that are wanted in the real world. Similarly, D is certainly not dependent on whether the machine is software or not. And there is nothing in the theory to limit in any way the nature of S.

We have used this theory, combined it with our practical experience, and produced a systematic requirements engineering method, REVEAL®. This method has been applied to a wide range of problems and we have not had any difficulty using the concepts outside a software context. We have used it, certainly, in conventional software development – for example to develop the requirements for a Certification Authority for multi-application smart cards. We have also used it in developing and analysing requirements for Railtrack's West Coast Route Modernisation Programme – an engineering project in which the key problems are not about software but about moving large numbers of people and large pieces of steel around the country reliably, fast and safely.

1.3 Benefits of Recognising the Commonality

There are important benefits if we recognise the commonality between systems and software requirements. Having a single requirements method is the most obvious one, but there are more subtle benefits. Each discipline has its own contribution to the requirements area and each can learn from the expertise of the other.

One of the main lessons that software engineers can learn from systems engineers is the importance of the application domain and the concept of operation. For example one of the main systems engineering standards, EIA/IS 731, includes activities such as "*Develop a detailed operational concept of the interaction of the system, the user, and the environment, that satisfies the operational, support, maintenance, and disposal needs.*" This emphasis on the operation of the system, and in particular on its interaction with its environment, is rarely found in the software requirements world.

On the other hand, software engineers are sometimes more comfortable with abstract notations than systems engineers are. Some of these notations are valuable in the requirements engineering activity. For example, we have found context diagrams to be extremely useful in any requirements project. They help to identify stakeholders and to discover necessary interfaces. Perhaps most important they act as a focus in one of the hardest parts of requirements engineering: determining the boundary of the system to be provided. It seems to me, therefore, that systems engineers should not condemn such diagrams, simply because they come from a software background, but should

look to these and other software-originated notations, to help in their task. Of course, it is a shame that the software world is abandoning context diagrams in its unthinking rush to embrace UML, but that is another matter.

1.4 Where does Software Fit in a System?

The main argument against the commonality of systems and software requirements is that (to paraphrase recent correspondence in the INCOSE requirements working group) "Software only comes in at the 7th level of decomposition". In other words, software requirements are seen as part of the detailed design of a much larger system.

However, this is purely a matter of perspective. Any system can be viewed as part of a larger system, and any system can be decomposed into smaller subsystems. At any level, a system interacts with its environment, whether that environment is the existing world or a collection of other subsystems within a larger system – or, most often, both.

Furthermore, there is in one sense a fallacy in the very idea of "software requirements". Very few so-called software requirements are really about the software at all: they are about the behaviour of the computer system as controlled by the software. This computer is clearly as much of a system as any other piece of hardware.

In some cases the "7th level" perspective is, indeed, appropriate. If we are thinking about the West Coast Route Modernisation, then the software that runs in the device that controls train speeds round curves does indeed exist in a constrained environment where it interacts with other components which are being designed as part of the same system. Furthermore the computer on which this software runs is embedded in a complex device which does other things – interacts with brakes, lineside equipment and so on.

On the other hand, there are plenty of systems where the software really is the system – rather than the software being there to control the computer, the computer is only there to run the software. An example is a flight data processing system (FDPS) for air traffic control: here, the hardware is entirely irrelevant and it is the manipulation of information which is the primary purpose of the system. It is true that. like every other system, the FDPS forms part of a larger whole, but in many cases the surrounding systems – radar, communications and so on – are pre-existing or at least not under the control of the FDPS project. In these circumstances, we can treat the development of FDPS as a system in its own right, and we should, for example, develop "*a detailed operational concept of the interaction of the system, the user, and the environment…*".

1.5 Conclusion

The theory of requirements engineering, and its practical application, are equally applicable to systems and software development. Systems and software engineers have in the past taken different approaches. While there will continue to be a need for different techniques and notations in the two fields, there is much to be gained by a common approach and a common understanding of the underlying issues.