What's the Use of Requirements Engineering?

Anthony Hall
Praxis Critical Systems

Abstract

There are many ideas about how to do requirements engineering and often they conflict with each other. Such conflicts can best be resolved by asking of anything one proposes to do: "What is the use of doing that?". The question demands a thorough understanding of the principles behind different methods, and the answers may surprise those who equate pragmatism with informality. I discuss how applying this rule helps in choosing requirements engineering methods and in dealing with the difficulties that arise in applying these methods.

1 Introduction

In engineering, the ultimate test of a theory is whether it is useful. There are many conflicting theories about what systems developers should do, about what methods they should use, and about how to apply particular methods. The conflicts arise not so much because one method is wrong and another right, but because they make different assumptions about what is useful. These assumptions are rarely explicit and in order to choose between methods we have to understand what their hidden assumptions are. They can often be revealed by asking, of any activity, "what is the use of doing this?" In this talk I will describe some of the methods we find useful for requirements engineering and analyze the characteristics of those methods by looking at their underlying assumptions. In particular I will examine our understanding of the nature of requirements and how to understand and resolve the conflict between the formal and the structured schools of analysis.

2 The World and the Machine

I make no apology for plagiarizing the title of Michael Jackson's invited talk to ICSE '95 [1]. For many years it has been received wisdom that describing the environment is a key aspect of requirements capture, but it was not until Parnas' work on the Four-Variable Model [2] and Michael Jackson and Pamela Zave's work published in 1995 that there was a clear exposition of what exactly this meant. The insights that this series of papers gave us are now fundamental to the way we define requirements.

I cannot overemphasize the importance of this framework. Applying it in practice has helped enormously to understand the problems and to structure our solutions. One of the many insights it has given us is the importance of the domain description. When systems fail, it can often be traced back to a poor or missing domain description, not to the requirements themselves.

It turns out, however, that deciding what is the domain and what is the machine is not always obvious. Frequently it is useful to think of "the machine" as not just the computer system we are building, but also many of the people who will interact with it. For example I recently worked on a safety-critical communications system. We could not guarantee, by the computer alone, that a user was speaking to the person they thought they were. So it was necessary to put in place not just the computer, but also a human protocol for checking that the correct parties were in communication. Conversely, the physical system we are building sometimes has to be regarded also as part of the domain. For example many operational systems have to monitor their own health. Parts of the machine may fail in ways over which we have no control. In general, we find it useful to regard as in the machine those things we can control, whatever form they take, and as in the domain those things we cannot control, even if they are what we are building.

3 The Role of Formality

We want to specify only those properties of the machine that affect the domain. However, we do want to make that specification as useful as possible. This means that the specification must be precise, because an ambiguous specification is bound to be misinterpreted. It must also be expressive, because we want to make the specifications as close to the users' conceptual model as possible, and to say, for example, what the system must not do as well as what it must do. We therefore use mathematical notation – formal methods – as the prime means of system specification. Mathematical notation is both precise and expressive.

There are, of course, some problems with current formal methods. An obvious difficulty is communication: few users understand the formal notation itself. The formal notation must for this reason (as well as many others) be accompanied by natural language or domain-specific notations. This is not, however, a difficult or problematic task. A more fundamental problem is the expressiveness of current formal notations. We have found it much easier to use formal notations for the system specification than we have for the description of the domain or for the statement of requirements. I suspect this is for two reasons. First, writing specifications is what formal notations were invented for. Second, the domain is typically vastly more complex than the machine, so to describe it formally would require a good deal of effort, effort that is not worth

1090-705x/97 $5.00 © 1997 IEEE

while unless the notation is well matched to the domain. This problem is mitigated by the fact that many domains have well understood languages of their own, but these are rarely fully formal. The validation of the specification against the domain properties therefore remains a weak link which can only be strengthened by a dialogue between domain experts and the engineers writing the specification. This is why, regardless of whether one is using a formal notation or not, it is never sufficient to take on trust a customer's written statement of requirements: it is always essential to talk to real system users in great depth. Elicitation and validation with users are therefore an integral part of our requirements engineering process.

Jackson has pointed out that different problems fall into different "problem frames": and a real problem, of course, never falls into a single frame. At the very least one needs to be able to conjoin several descriptions of the same problem. Only a few languages, of which Z is one, allow this freedom. Even that is not enough, however, since some aspects of problems – for example user interfaces – cannot easily be expressed in Z, and so we need to combine different languages in the same specification. This is currently an active area of research. We have often had to combine different notations but we do not yet have a good theory of how to do it.

I do not want to overemphasize the problems: on the contrary, we find formal methods extremely useful in practice. An example is CDIS [3], an ATC information system we built where we found that using formal specification reduced the number of defects in the delivered system. Furthermore the defects that remained were very rarely specification errors. This suggests that formal specifications do indeed help both to build the system right and to build the right system. This improvement was free: CDIS cost no more to build than it would have with conventional methods.

4 Nonformal Approaches

We do not use only formal methods in requirements engineering. On the contrary, we use notations and ideas from structured and object-oriented analysis. However, only some of the ideas of conventional analysis are useful, while others are positively harmful. It is important to understand why that is so.

A typical structured analysis starts with a "context diagram", which shows the system and all the external entities that interact with it. The requirements are then defined by decomposing the system into a number of processes and defining the inputs and outputs of each process. These processes in turn are broken down, until elementary processes are reached and these are then defined using some other notation such as pseudocode. The initial step here is entirely right, and we always draw a context diagram as a step in defining the machine and the domain and in clarifying the boundary between them. From then on, however, structured analysis is not addressing requirements at all – it is sketching a design of the system. The "processes" in the analysis do not correspond to anything real in the domain.

An alternative, more fashionable approach is to use object-oriented analysis. Here instead of using processes as the fundamental units one uses objects. The first step in OOA is to identify the objects that exist in the domain. This, too, I consider a vital part of defining the domain and an object model always forms part of our domain description. Again, however, one can fall into a trap: one can try to define the behaviour of the system in terms of the behaviour of the individual objects, typically treating each object as a state machine. Once again, this is carrying out design, not specification: it can be extremely difficult to understand the behaviour of a system if one is only given the behaviour of its components. There are OOA methods like Fusion which do not fall into this trap: that is because the basic ideas of such methods, if not the notations they use, are the same as those of formal methods.

The common theme to these examples is that describing how a system might be built is not a good way to specify what it does. The hidden assumption in methods where the analysis model is like a design is that such a design is a good model of the system, either because it makes it comprehensible to the users or perhaps because it advances the project towards its goal of an implemented system. We have found, on the contrary, that separation of concerns between the users' conceptual model and the physical design is essential. The characteristics which make a structure comprehensible to a user are completely different from those which make it a good design.

5 Using Requirements

Good requirements are essential if we are to be sure that we are building the system the users want, and that we are not doing more than is needed. They should, of course, continue to play this role not just when the system is initially built, but through its subsequent maintenance and enhancement. In practice there are very few systems where the requirements are good enough to be useful throughout the system's life. All too often the implemented system "supersedes" the requirements: a euphemism for saying that it does not meet its requirements, which were probably not truly requirements in the first place. It is only by a clear understanding of what a requirements definition should be, a clear separation between specification and design and the use of precise specification notations that we can make requirements definitions which are of lasting use.

References

[1] Michael Jackson. The World and the Machine. In *Proceedings,* 17^{th} *International Conference on Software Engineering*, 1995, pp. 283-292.

[2] D. L. Parnas and J. Madey. Functional Documentation for Computer Systems Engineering (Version 2). Technical Report CRL 237. Telecommunications Research Institute of Ontario, McMaster University, Hamilton Ontario 1991.

[3] Anthony Hall. Using formal methods to develop an ATC Information System. *IEEE Software,* March 1996, pp. 66-76.