Requirements Eng (2001) 6:132–134


Requirements
Engineering

*Viewpoints*

'Viewpoints' is a regular section in *Requirements Engineeering* for airing readers' views on requirements engineering research and practice. Contributions that describe results, experiences, biases and research agendas in requirements engineering are particularly welcome. 'Viewpoints' is an opportunity for presenting technical correspondence or subjective arguments. So, whether you are a student, teacher, researcher or practitioner, get on your soapbox today and let us know what's on your mind...

Please submit contributions electronically to Viewpoints Editor, Didar Zowghi (didar@it.uts.edu.au). Contributions less than 2000 words in length are preferred.

# Software Requirements Engineering: The Need for Systems Engineering and Literacy

**Roel Wieringa**
Department of Computer Science, University of Twente, Enschede, The Netherlands

## 1. Introduction

This viewpoint continues the favourite comparison between house engineering and software engineering, recently brought forward again by Dan Berry [1]. In the past eight years, I had two houses renovated and a third one built from scratch, which, after delivery, I had extended immediately by employing a different builder. Each of these four (rebuilding processes involved one main contractor and up to 10 subcontracting builders, who dealt with electricity, plumbing, painting, tiling, plastering, the central heating system, concrete floors, parquet floors, stairs, the kitchen, curtains, sun screens, etc. Some of these processes also involved negotiations with independent suppliers in the Netherlands and Germany, each with different trade laws. Financially this is not a very attractive sequence of moves, not to speak of the stress that this caused in agendas that are already overburdened, but a great opportunity to learn first hand about house design and implementation.

## 2. The Moral

I indicated three phenomena that also appear in software requirements engineering: First. I did not build or rebuild these houses myself, I *had* them built. This created a need for communication with the builders, a process in which house builders do not seem to fare better than software builders. As a consequence, requirements engineering sometimes continued until after the product (a house) was delivered. Second, with so many parties involved, a need for coordination arose that I have come to believe is the most important function of systems engineering to ensure success, i.e. user satisfaction, of the building process and product. Third, I was going to *live* in these buildings. In three cases, I already lived in them during the rebuilding process; in the fourth case, I lived somewhere else and had to frequently interrupt my work to check that what was being built was what I wanted. Likewise for software: software creates a semantic universe in which people will live for an important part of their lives. The fact that this universe consists of concepts, and is therefore physically invisible, means that it is even closer to us than the physical world in which we live. Software becomes part of our minds because it changes the concepts in terms of which we think. Like the evaluation of the quality of a house, evaluating the quality of software has an important emotional aspect that engineers and builders know exists but that they cannot very well deal with. They are selected for their jobs because they know how to deal with devices, materials, or software, not because they know how to deal with people. House users deal

*Correspondence and offprint requests to:* R. Wieringa, Department of Computer Science, University of Twente, PO Box 217, 7500 AE The Netherlands. Email: roelw@cs.utwente.nl

with unsatisfactory properties of their house by fiddling around with the house *after* delivery for several years, until it fits like an old shoe (and perhaps looks like one). Most software users are less fortunate since they have to live with what is delivered to them, adapting to the software where it does not fit their needs.

But I am getting ahead of my story. Let me first recapitulate the similarities between house engineering and software engineering.

## 3. The Story

First, all specifications are incomplete, and necessarily so: a complete house design would be as big as a house. Second, what *is* specified is often not understood by the user, even if he thinks he understands. Third, what is specified is not always what is built: many builders do not like to read the specification but simply build what they think is normal; which often is not what we specified. At one time, we found some of the builders that did use a specification were using the specification of the neighbour's house. In their daily routine, they had forgotten to check the specification identifier. We have also caught builders using an outdated version of the specification – how are they to know there is a more recent one? Fourth, what is specified sometimes *cannot* be built. We found that when this happens most builders prefer to solve the problem themselves. Those who do discuss the problem with the user talked to us in impenetrable jargon that made the problem seem much more difficult than it was. Fifth, of course we could not state all our requirements up front. Seeing the house take shape gave us so many ideas that we regularly updated the specification – at a cost that seemed to rise exponentially with time. Dan Berry suggests this is a good strategy from a builder's point of view but I can assure you that it does not increase user satisfaction. Trying to increase our satisfaction by improving the specification, we became disgruntled by the price that we had to pay for the update. This does not help to build a good customer–producer relationship, which in turn does not facilitate dealing with the numerous small problems that the builder, by contract, should solve after the house is delivered. Sixth, each subcontractor has its own view of the product. These views must be consistent, and the work performed according to them must be coordinated. The dependencies between these views turned out to be circular – nothing could be decided until everything was decided – and they created interference between deadlines. If planning became difficult because of this, it became nearly impossible due to turbulence in the project environment, such as an excessive amount of rain, shortage of manpower, and theft from the building site.

So what can we learn from this? First, that house building is an infinite source of dinner table stories, or better still, strong drinking stories for your buddies. Second, we can learn some lessons about building systems and building their specifications.

## 4. Building Systems

A system is a coherent collection of elements that interact so that useful properties for the environment may emerge. The keyword here is coherence. To achieve coherence, all subcontractors, and the views they have on the system, need to be coordinated with each other. To achieve that the elements of the system jointly deliver a useful service; they need to be aligned with the user requirements. This is exactly the province of systems engineering, which encompasses a mix of requirements engineering and project management all aimed at achieving coherence and the emergence of useful properties of the system [2]. The above story illustrates that bad house engineering is very similar to bad software engineering. Both are examples of bad systems engineering. Our house-building processes could obviously have been improved by using systems engineering principles for managing changing requirements, updating the specification, managing subcontractors, and coordinating different views on the system.

## 5. Building Textual Information

Although house engineering and software engineering are agreed to be examples of systems engineering, they differ in the fact that software consists of textual information and houses do not. Software is a set of instructions written for a machine to execute. This entails other well-known differences, such as (1) software is invisible because it is conceptual, (2) it seems easily changeable and (3) there is no clear break between specification (writing text) and building (writing more text!), as there is in all other branches of engineering. A further consequence is that (4) the product behaves exactly as we programmed. This differs from the kind of engineering where we put together physical material so that the resulting system of materials has certain desirable properties. These materials are not *defined* by our specifications but *described* by them. They do not, by definition, behave exactly as we programmed. They behave as physical laws tell them to do, not as we tell them to do. A final consequence is that (5) software products are symbol-manipulating products. They become part of the user's semantic universe. This means that software systems

engineering becomes a conceptual exercise, not only in its requirements engineering part, but also in its job of coordinating stakeholders, specifications, views and building activities. It is all right for a house builder to fail to conceptualise a problem he encountered. He did not have the training to do this, nor is there a need to do it, because he can physically show what the problem is. Software system engineers *should* have the training to conceptualise construction problems in advance during discussions with the user, because software is conceptual and there is no other way to discuss problems with the user.

## 6. Requirements Engineering Education

A consequence of these observations is that requirements engineering education should not only contain a course on systems engineering, but also a course on technical writing. Current courses on technical writing explain how to write technical documentation and user manuals but do not deal with writing requirements. Kovitz [3] is an exception to this, but then this is a book about requirements, not about technical writing. More should be done in this area.

Requirements engineers should be aware of the fact that person-to-person communication contains a limitless number of unspoken assumptions, some of which may have to be uncovered during the conversation. This is relevant when the conversation is about a product design, because it is one of the mechanisms of requirements evolution. Much of the activity of engineering consists of communicating about a product between people from different disciplines. Systems engineering provides a means of communication across disciplines, and requirements engineering should likewise deal with unspoken assumptions and desires and make them explicit. Where this requires good technical writing skills for any kind of requirements engineering, in the case of software requirements engineering it requires additional skills in combining clarity with precision of expression. In software requirements engineering, language is not only the means of communication with other people, but also the basis for a program that must be executed by a machine. That leaves no room for vagueness. Technical writing skills are more essential to good software requirements engineering than skills in formal specification or skills in using UML diagrams and structured analysis.

## References

1. Berry DM. Software and house requirements engineering: lessons learned in combating requirements creep. Req Eng 1998;3:242–244
2. Stevens R, Brook P, Jackson K, Arnold S. Systems engineering: coping with complexity. Prentice-Hall, Englewood Cliffs, NJ, 1998
3. Kovitz BL. Practical Software requirements: a manual of content and style. Manning, 1998