How much Requirements Engineering do we need?[1]

Martin Glinz
University of Zurich, Switzerland
glinz@ifi.uzh.ch

Abstract

As unambiguous and complete requirements specifications are not feasible in most cases, we investigate the question how much Requirements Engineering (RE) we actually need for the development of successful systems and products.

Based on the notion of value of requirements, we discuss risk, shared understanding and customer-supplier relationship as major influencing factors.

1. Motivation

Until recently, there was a kind of dogma in Requirements Engineering (RE) that any good requirements specification needs to describe the requirements comprehensively. This is reflected in the classic quality attributes of completeness and unambiguity, which are typically considered as crucial qualities of a requirements specification [6], [7], [8].

However, the advent of agile development is now seriously challenging this dogma. Advocates of extreme agilism even recommended to abandon requirements specifications, use some stories or feature lists instead, and fully concentrate on programming [1]. In the meantime we have learned that such an extreme attitude to requirements only works under favorable conditions. Otherwise, it typically leads to unsystematic hacking or even into disaster. So there is a need to explore the middle ground more systematically: how much RE do we really need for the successful development of systems and products?

In this short paper, I try to contribute to the clarification of this question by looking at some factors that influence how much RE we need. In particular, I will discuss the factors of risk, shared understanding, and customer-supplier relationship.

2. Unambiguity and completeness as properties of a good requirements specification

In this section we briefly discuss the classic notion of unambiguity and completeness as core characteristics of a good requirements specifications. We start with some quotes. The IEEE 830 standard back in 1984 stated in the foreword: "(...) the result of the software requirements specification process is an unambiguous and complete specification document" and continues on page 11: "A good SRS is: (1) Unambiguous, (2) Complete (...)" [5]. In the 1998 revision of this standard we find on page 4: "An SRS should be a) Correct; b) Unambiguous; c) Complete; (...)" [6]. The IREB foundation level syllabus [7] states on page 16: "(...) the requirements document must meet certain quality criteria. In particular this includes: Unambiguity and consistency (...) Completeness (...)". That sounds good. However, are these qualities actually achievable or are we chasing a fiction?

Let's look at unambiguity first. An unambiguous requirements specification requires a formal specification language as every statement needs to have exactly one meaning. We all know about the very limited success of formal specification languages in practical RE, but let's assume we had a suitable formal language. In this case, an unambiguous specification would be feasible for software that is of type S in Lehman's classification [9]. The type S in this classification encompasses all software that solves a precisely specifiable problem, without referring to any real world phenomena. For example, a program for sorting a set of integers is of type S. However, for any real world software (the type E in Lehman's classification), we are confronted with the problem of mapping phenomena of the real world on constructs of our specification language. This is a modeling process which, by its very nature, is not formalizable. Hence, while the statements in the modeling language can be made unambiguous, the mapping that leads to these statements can't. So, eventually, a 100% unambiguous description of requirements in the real world is impossible.

Next, we look at completeness. The ISO-IEC-IEEE 29148-2011 standard [8] says about completeness: "Complete. The set of requirements needs no further amplification because it contains everything" (p. 11) and "The SyRS should completely describe all inputs, outputs, and required relationships between inputs and outputs" (p. 43).

As in the case of unambiguity, this notion of completeness works for software of type S according to Lehman's classification [9]. For the problem of sorting a set of integers mentioned above, describing all inputs and outputs is obviously feasible and makes sense. However, what about type E software, which is embedded in a real world environment and constitutes the bulk of today's software: is it possible there to describe all inputs, including everything that could ever happen even in the most improbable case? The answer is no. As our perception of the world is limited, we will never be able to describe all inputs in terms of everything that could ever happen in the real world.

[1] Extended version of a keynote talk given at the meeting of the GI Fachgruppe Requirements Engineering, Nov 26-27, 2015 at FHNW in Windisch, Switzerland. The keynote was held in German under the title «Wie viel Requirements Engineering braucht der Mensch?». Accordingly, the style of this paper is more colloquial than usual.

Moreover, even if we were able to create a truly complete and unambiguous requirements specification, it would not make sense to write it in most cases, as it would cost too much. We have to keep in mind that requirements are a means, not an end.

3. What now?

The fact that a complete and unambiguous requirements specification in fact is not achievable, puts the whole idea of full-fledged requirements specifications into question. Extreme approaches such as XP [1] that do away with requirements specifications altogether, throw out the baby with the bathwater – they only work under specific favorable conditions, where the embedded customer knows everything about the requirements without needing to elicit requirements or document them for the purpose of communication.

As neither of the extremes is actually feasible in the general case, we need more in-depth reflection about how much effort we should invest in RE.

However, determining the right amount of requirements specifications is not easy. As an illustration, consider the following example.

Example. Let's assume a provider of training courses needs a new course administration system. For the participant entry form, the following requirement has been specified: "The participant entry form shall have fields for the participant's *name*, *first name*, *sex*, and *address* and a submit button." Does that suffice? When the supplier and the customer know each other and closely collaborate, it surely would. When the development is outsourced to a low-cost organization with no domain knowledge, a form such as the one shown in Fig. 1 might result, which obviously would not satisfy the stakeholders.

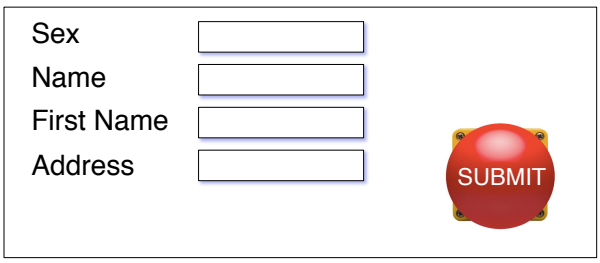

Figure 1: A potential result when not specifying enough

Our core idea for addressing the question of how much RE we need is to consider the *value* of requirements. The value of a requirement can be conceptualized as the *benefit* of a requirement in terms of reduced development risk minus its *cost* [3]. Eventually, we specify requirements because we want to reduce the development risk, i.e., the risk of developing a system that does not satisfy its stakeholders' desires and needs. If that risk is low or even zero, specifying requirements explicitly is indeed a waste of effort. The cost of a requirement, on the other hand, is the cost of eliciting, documenting and maintaining it.

4. Three major influencing factors

Among many factors that influence how much we should invest into RE, we briefly discuss three key factors in this paper: risk, shared understanding, and customer-supplier relationship.

4.1 Risk

The notion of requirements value leads to the question how we can assess the contribution of a requirement to the reduction of development risk. A first means is a categorization of criticality according to stakeholder importance and impact (Fig. 2) [3]. The more important the stakeholder(s) of a requirement and the higher the impact when a requirement is missed, the higher is the development risk associated with that requirement. Table 1 (partially adapted from [3]) lists further factors that influence the risk and, hence, the amount of explicit specifications needed.

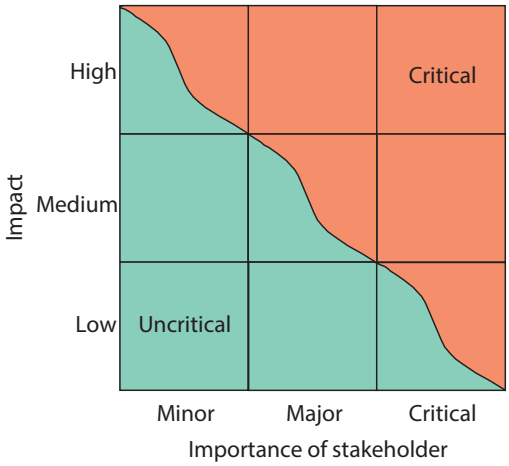

Figure 2. Assessing the criticality of a requirement [3]

Table 1. Factors influencing the risk and value assessment

Criticality	The more critical a requirement, the more its explicit specification contributes to the reduction of development risk.
Distinctive ness	When a requirement is distinctive for a product to be developed, not meeting such a requirement has a high impact on the product. So, a higher effort for specifying this requirement is justified than for non-distinctive requirements.
Shared understanding	The better the implicit shared understanding about a requirement, the less explicit specification we need for controlling the development risk.
Reference systems	When a reference system (for example, a previous product or a competitor's product) exists, referring to that reference system reduces misunderstandings and, hence, the risk.
Length of feedback cycle	The shorter the feedback cycle (the time interval between expressing a need and receiving a system that should satisfy this need), the lower is the development risk.
Customer-supplier relationship	The better the customer-supplier relationship (in terms of mutual respect, understanding, and trust), the less requirements need to be contractually specified for achieving low development risk.
Certification required	If an authority needs to certify a system, all certification-relevant requirements might need to be specified explicitly and in detail, regardless of the development risk.

4.2 Shared understanding

Shared understanding between stakeholders and software engineers is a crucial prerequisite for successful development of software [2], [4]. The notion of shared understanding has two facets [4]: *explicit* shared understanding is about interpreting explicit specifications in the same way by all people involved. *Implicit* shared understanding denotes the common understanding of knowledge, assumptions, opinions, and values that have not been specified explicitly.

Writing requirements specification documents aims at explicit shared understanding of the problem and the requirements associated with it. Accordingly, the question of how much RE we need translates into the question to what extent we can rely on implicit shared understanding of requirements instead of documenting them explicitly. While unreflected reliance on implicit shared understanding can be dangerous, deliberately building and assessing implicit shared understanding is a viable alternative to explicit specifications in many cases, as it costs less while achieving a similar degree of shared understanding. Details may be found in [4].

The implementation of the *sex* feature as a text field in the form shown in Fig. 1 is a typical example where shared understanding was assumed, but failed. The specifiers believed that using checkboxes or radio buttons for implementing this feature is common knowledge. So they did not explicitly specify that further, while the programmer who coded the form did not know about this convention and chose just the cheapest and fastest way of coding this feature.

The extent to which one can rely on implicit shared understanding depends, among other factors, on shared values and a suitable customer-supplier relationship (see Sect. 4.3 below). Implicit shared understanding must not be used in a naive, unreflected way. Instead, it has to be built and assessed in order to become dependable [4]. Nevertheless, this can be still much less effort than trying to specify everything explicitly.

4.3 Customer-supplier relationship

The relationship between the supplier of a software system and its customer(s) also has a strong influence on how much RE we need. For example, when the development of a system is outsourced and/or the customer and the supplier do not trust each other, detailed explicit specifications are crucially needed in order to keep the development risk low. Similarly, factors such as domain knowledge of developers, length of feedback cycles or geographic distance can have a strong influence on how much RE we need. The example given in Sect. 3 above illustrates that the very same requirement might be sufficient in a perfectly working customer-supplier relationship, while it may lead to disastrous results such as the one shown in Fig. 1 in case of a dysfunctional customer-supplier relationship.

5. Conclusions

Starting from the problem that a complete and unambiguous requirements specification is neither possible nor economically feasible in most cases, we have investigated the question how much RE we actually need. We have shown that the notion of value of requirements plays a core role and have briefly discussed three factors that influence that value and, to a large extent, determine how much RE is optimal in a given situation.

References

[1] Kent Beck (2004). *Extreme Programming Explained: Embrace Change*. 2nd edition, Boston: Addison-Wesley.

[2] Eva A.C. Bittner, Jan M. Leimeister (2013). Why Shared Understanding Matters – Engineering a Collaboration Process for Shared Understanding to Improve Collaboration Effectiveness in Heterogeneous Teams. *Proceedings Annual Hawaii International Conference on System Sciences*. 106–114.

[3] Martin Glinz (2008). A Risk-Based, Value-Oriented Approach to Quality Requirements. *IEEE Software* **25**(2):34–41.

[4] Martin Glinz, Samuel Fricker (2015). On Shared Understanding in Software Engineering. *Computer Science – Research and Development* **30**(3-4):363–376.

[5] IEEE (1984). *IEEE Guide to Software Requirements Specifications*. IEEE Standard 830-1984. IEEE Computer Society Press.

[6] IEEE (1998). *IEEE Recommended Practice for Software Requirements Specifications*. IEEE Standard 830-1998. IEEE Computer Society Press.

[7] IREB (2015). *IREB Certified Professional for Requirements Engineering - Foundation Level - Syllabus* Version 2.2. https://www.ireb.org/content/downloads/2-syllabus-foundation-level/ireb_cpre_syllabus_fl_en_v22.pdf. Visited 2016-01-27

[8] ISO/IEC/IEEE (2011). *Systems and Software Engineering — Life Cycle Processes — Requirements Engineering*. ISO/IEC/IEEE Standard 29148, First edition 2011-12-01.

[9] Meir M. Lehman (1980). Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE* **68**(9):1060–1076.