Chapter 3
Problem and Solution Concepts in Requirements Engineering

Requirements engineering designates both the rigorous practice of requirements problem solving and the field of research which studies this practice and ways to improve it. This chapter connects the ideas discussed in Chaps. 1 and 2 to the basic ideas and terminologies of requirements engineering. This is important because various requirements modeling languages and algorithms, that is, AI for requirements problem solving, have been proposed in requirements engineering since the origin of the field in the 1970s.

3.1 Requirements Engineering

This section suggests the following relationship between requirements problem solving and requirements engineering: The former designates the phenomenon which the latter studies and aims to influence.

Requirements engineering is a term that designates both an engineering discipline and a field of scientific research.

The engineering discipline covers various activities, such as elicitation, modeling, analysis, negotiation, and so on, which are carried out in order to define rules that a system has to satisfy when it is made and used. The rules can originate in expectations of stakeholders who invest, use, or otherwise influence and are influenced by the system, in the conditions of the system's operating environment, its regulatory environment, and so on.

It is an engineering discipline. These activities have to be done rigorously, in planned steps, using tried and tested mathematical or other tools.

Requirements engineering as a field of scientific research studies a variety of topics, such as information elicitation [38, 55, 70], categorization [36, 84, 153], vagueness and ambiguity [81, 96, 105], prioritization [11, 69, 88], negotiation [13, 85, 95], responsibility allocation [23, 36, 50], cost estimation [14, 17, 128], conflicts and inconsistency [68, 107, 142], comparison [96, 97, 105], satisfaction evaluation [16, 90, 105], operationalization [46, 50, 53], traceability [31, 56, 115], and change

I. Jureta, *The Design of Requirements Modelling Languages*,
DOI 10.1007/978-3-319-18821-8_3

[20, 26, 146]. Each topic is related to issues and tasks that occur during requirements problem solving.

Historical origins of requirements engineering are in software engineering, and specifically in the challenge to define and document what the system should do for its stakeholders and in its environment, without saying exactly how it will do this. The "how" usually remains outside the scope of requirements engineering and is the responsibility of those engineering, making, maintaining, and changing the system.

The notion of *system-to-be* is a term that emphasizes it is not made yet, or simply *system* is central to requirements engineering.

Due to the historical origins, system usually designates *software and hardware*. At its boundary are the people who interact with it and any other thing in the environment which the system can somehow exchange information with, influence, and be influenced by.

The phenomenon that the requirements engineering discipline and field focus on existed before either the discipline or the field was formally recognized. Requirements engineering arose in response to situations observed in systems engineering in general, of not knowing how to make sure that the system being made will in fact appropriately address the issues that motivated making it in the first place.

Requirements problem solving designates that phenomenon, namely all that people do when they have unclear, abstract, incomplete, and potentially conflicting information about expectations of various stakeholders and about the environment in which these expectations should be met; a system should be made to satisfy these expectations, and they want to define rules, such that if the system is made to satisfy these rules, then it will also satisfy the expectations in its given environment.

Requirements problem solving is present when designing new and changing existing systems. It needs to be done for any system class and domain, and regardless of the extent to which people are involved in the system, from autonomic Internet-scale clouds, to traditional desktop applications, industrial expert systems, and embedded software, all enabling anything from massive mobile application ecosystems, global supply chains, medical processes, business processes, mobile gaming, and so on. Requirements problem solving is done regardless of how the software in the system is designed and made, from a military waterfall approach to a startup's own agile dialect, and from organizations where software engineers talk directly to customers, to those where product designers, salespeople, or others mediate between requirements and code. In all these cases, there will be unsatisfied expectations and the need to make systems to satisfy them.

3.2 Problem and Solution

This section introduces definitions for the terms problem and solution, and relates them to the notions of problem situation, solution situation, and requirements problem solving.

Problem and solution are common terms. The dictionary definition of problem is that it designates "a matter or situation regarded as unwelcome or harmful and needing to be dealt with and overcome." The corresponding definition for solution is that is "a means of solving a problem or dealing with a difficult situation."[1]

This section introduces definitions for "problem" and "solution" which are specific to this book. They are simple, uncontroversial, and coherent with their dictionary definitions. The main benefit of having specific definitions is that they use the terminology introduced for requirements problem solving in Chap. 1. Secondary benefits are less obvious, and I will highlight them below.

3.2.1 Problem

I use the term "problem" to refer to *ideas* about what is observed or believed to be true in a problem situation. Problem is what you observe or think is true in the problem situation. It is *not* a record of these ideas, for example, a document where you wrote them down. It is the ideas or thoughts themselves.

It is important that a problem designates ideas, not their representations. This is because I argued earlier in Sect. 1.2 that different people can see different problems in the same situation. They may pay attention to different events, things, and individuals. They may draw different conclusions about what is and is not desirable in that situation. You may hold one set of ideas, but there is no reason others should share them.

Instead of using the terms "ideas" and "thoughts" in my definitions, it is more conventional in requirements engineering to talk of *propositions*. I consequently say that you and I may believe different *propositions* to be true of a situation, even if we are in that same situation. The term "proposition" has a specific definition in philosophy, and I follow the one from Matthew McGrath [101] in the Stanford Encyclopaedia of Philosophy:

> *"Propositions […] are the sharable objects of the attitudes and the primary bearers of truth and falsity. This stipulation rules out certain candidates for propositions, including thought- and utterance-tokens, which presumably are not sharable, and concrete events or facts, which presumably cannot be false."*

Tying the above to problem situation leads me to the following simple definition for the term "problem."

Definition 3.1 *Problem*: propositions believed to be true of a problem situation.

[1] Both quotations come from a Google search for keywords "define:problem" and "define:solution."

3.2.2 Solution

Comments I made for the term "problem" apply for the term "solution." Solutions are ideas believed to be true of the solution situation. Hence the following definition.

Definition 3.2 *Solution*: propositions believed to be true of a solution situation.

The major difference from the common sense definition of the term "solution" is that here, "solution" is not that which brings about the solution situation. It amounts to propositions about the solution situation. As I explain in Sect. 3.3, I use the term system for that which brings about the solution situation.

3.3 System

This section introduces the term system and relates it to the terms introduced so far.

Although the historical origins of requirements engineering are in software engineering, the term "system" in contemporary requirements engineering is not restricted only to software and/or hardware. Its scope can include only limited to specific (parts of) software and hardware, or widened to include such issues as work guidelines, business processes, responsibilities, contracts, or other concerns.

As various things can be part of a system, I prefer not to define the term by saying what can be in it, or has to stay outside. It makes no difference in this book what exactly is, or is not part of a system. What matters is that the system is all that is made and used to bring about a solution.

Definition 3.3 *System*: that which is made and used in order to make solution true.

A system need not be about software or hardware. It can be a brand, a political election program, a corporate strategy, or a business process.

In all cases I discuss in this book, the system is not restricted to software and hardware. In some of the cases, software and hardware were not mentioned at all as important parts of systems which were actually used.

3.4 Model

This section introduces the terms model, problem model, solution model, and relates them to those introduced so far.

The remaining piece of the puzzle is to describe solutions, problems, and systems in such a way that we can communicate about them during requirements problem solving. This is done with models.

Definition 3.4 *Model*: representation of propositions.

This is not a conventional use of the term model in requirements engineering. Model is normally used to designate the representation of the system only. However, I need to talk about representations of solutions, problems, and systems, which lead me to several kinds of models.

Definition 3.5 *Problem model*: representation of a problem.

Definition 3.6 *Solution model*: representation of a solution.

Definition 3.7 *System model*: representation of propositions believed to be true of the system.

It is on the basis of system model that the system is implemented, updated, changed, its new releases planned, made, announced, and rolled out. The system model's scope may be limited to specific (parts of) software and/or hardware, or widened to include such issues as work guidelines, business processes, responsibilities, incentives, contracts, or other concerns.

System model can take different forms, from minimalistic to-do lists that hint at stakeholders' expectations and subsume implicit design and engineering solutions, to elaborately structured documentation on contracts with employees and suppliers, responsibilities of positions in the value chain, guidelines for employee coordination and collaboration, as well as software pseudocode.

A system is not the output that requirements engineering produces. Requirements engineering does not include, for example, the detailed engineering, development, testing, release, maintenance, and so on, of software that may be part of the system, nor can it include the training of people who should use it. In the law firm owner's case, the solution included changes in contracts with employees, in incentives, training, team building, among others. They are activities which in order to be done well require specific expertise and are delegated to those who have it.

Problem models, solution models, and system models are the output sought in requirements engineering and requirements problem solving.

3.5 Default Problem and Solution

This section presents and discusses the default problem and default solution concepts in requirements engineering.

There is a default definition of the problems that requirements engineering tries to solve when applied. There is also a default definition of the solution sought. It is important to know them because they highlight a number of assumptions made in requirements engineering.

The de facto default view in requirements engineering is that requirements problem solving is done incrementally, starting from incomplete, inconsistent, and imprecise information about the requirements and the environment, and that each

design step reduces incompleteness, removes inconsistencies, and improves precision, toward the system model [15, 23, 36, 46, 48, 58, 80, 107, 119, 141, 153].

This general view of the problem-solving process, which you start with less detailed and somehow deficient information, and then increase detail and remove deficiencies, is also shared in other domains involving design, such as architecture [92, 134] and civil engineering [4].

Within that view, which requirements engineering has of requirements problem solving, what is the default definition of problems and solutions?

The most influential treatment of this question is in Pamela Zave and Michael Jackson's seminal paper "Four Dark Corners of Requirements Engineering" [153], and is echoed in discussions on the philosophy of engineering [132]. Their view is aligned with some of the most influential research in requirements engineering, which both preceded and followed said paper. This includes, for example, contributions from Boehm et al. [13, 15], van Lamsweerde et al. [36, 37, 96, 141, 142, 143], Mylopoulos et al. [23, 58, 107], Robinson et al. [119], Nuseibeh et al. [74, 107], to name some.

According to Zave and Jackson, requirements engineering is successfully completed in any concrete engineering project when the following conditions are satisfied [153]:

1. *"There is a set R of requirements. Each member of R has been validated (checked informally) as acceptable to the customer, and R as a whole has been validated as expressing all the customer's desires with respect to the software development project.*
2. *There is a set K of statements of domain knowledge. Each member of K has been validated (checked informally) as true of the environment.*
3. *There is a set S of specifications. The members of S do not constrain the environment; they are not stated in terms of any unshared actions or state components; and they do not refer to the future.*
4. *A proof shows that $K, S \vdash R$. This proof ensures that an implementation of S will satisfy the requirements.*
5. *There is a proof that S and K are consistent. This ensures that the specification is internally consistent and consistent with the environment. Note that the two proofs together imply that S, K, and R are consistent with each other."*

Using the terms I introduced so far, Zave and Jackson's conditions translate as follows:

1. There is a requirements model, call it R, which stakeholders agreed on. It represents propositions that convey stakeholders' expectations.
2. There is an environment model, called K, which stakeholders agreed on. It represents propositions believed to be true of the environment in which the system will run.
3. There is a system model, call it S, which describes propositions true of the system.
4. If the propositions represented in the environment model are true, that is, the environment is as described, and the system is made and runs in that environment according to the system model, then propositions described in the requirements model will also be true.

5. If the system is made and runs according to system model, then the environment will remain as described in the environment model, and if the environment remains as described, then the system will continue to run without violating system model.

The translation emphasizes that there are *representations* of three kinds of propositions, namely requirements, domain knowledge, and system propositions. The translation also does *not* assume that any of these representations is written in classical logic, and therefore, cannot talk about proofs. Instead, it rewrites the fourth and fifth conditions without assuming the language used to make the representations. All these are minor changes, and the translation preserves the central ideas.

Perhaps the most important observation to make about the conditions from Zave and Jackson is that they do not talk about the structuring of the problem and solution, and about the design of the system. In other words, there is no indication that this is ill-structured problem solving. The conditions should be checked after the requirements, environment, and system are clear enough, to make problem solving well structured.

Returning to the main topic of this section, the translation suggests a default problem and solution for requirements engineering.

Definition 3.8 *Default problem*: there are

1. a set R^p of requirements propositions, which are propositions believed to be true of what stakeholders expect, and
2. a set K^p of environment propositions, which are propositions believed to be true about the environment in which the system will be used,

and it is not sufficient for the environment propositions alone to be true, in order for requirements propositions to be true.

The default problem is that you know something about stakeholders' expectations and about the environment in which they need to be satisfied, yet that environment alone does not ensure that these expectations indeed are satisfied.

I write X^p for a set of propositions, and X for the set of representations of propositions, which may, but need not be related to those in X^p. The reason I dissociate X from X^p is that it is hard to be sure that all propositions in R^p are accurately represented by the content of R. Keep in mind that propositions in R^p are ideas, not representations of ideas. They are hard to access, so to speak, because by being ideas, they are "in the mind" of your own and of others. Going from R^p to R is complicated, involves having people communicate with you during elicitation about propositions, and thus probably means that you and someone else would produce different R sets, from presumably the same R^p.

In contrast to the default problem, the solution is not the input, but the output of problem solving, which comes therefore after problem structuring, and is about models.

Definition 3.9 *Default solution*: there are

1. a requirements model R, which stakeholders agreed on, and which may represent propositions from R^p in the Default problem,
2. an environment model K, which stakeholders agreed on, and which may represent propositions form K^p in the Default problem,
3. a system model, call it S, which describes propositions true of the system,

and the system model S is such that

1. if the propositions represented in K are true, that is, the environment is as described, and the system is made and runs in that environment according to the system model, then propositions represented in the requirements model R will also be true, and
2. if the system is made and runs according to S, then the environment will remain as described in the environment model K, and if the environment remains as described in K, then the system will continue to run without violating the propositions represented in S.

Keep in mind that K in the default solution does not need to represent exactly all or any of the propositions in K^p in the default problem. Same applies to requirements propositions in R^p and the requirements model R. This is because the default problem triggers requirements problem solving, which involves problem structuring, and the information known for the original problem can be removed or replaced.

http://www.springer.com/978-3-319-18820-1

The Design of Requirements Modelling Languages
How to Make Formalisms for Problem Solving in Requirements Engineering
Jureta, I.
2015, XII, 286 p., Hardcover
ISBN: 978-3-319-18820-1