Handbook of Requirements and Business Analysis

Bertrand Meyer

Handbook of Requirements and Business Analysis

Springer

Bertrand Meyer
Schaffhausen Institute of Technology
Schaffhausen, Switzerland

ISBN 978-3-031-06738-9 ISBN 978-3-031-06739-6 (eBook)
https://doi.org/10.1007/978-3-031-06739-6

Cover illustration: “Émailleur à la Lampe, Perles Fausses”, plate from Diderot’s and D’Alembert’s Encyclopédie, by kind permission of the ARTFL Project at the University of Chicago.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Short contents

The full table of contents starts on page xix.

Preface

If a system is a solution, requirements state the problem. Since a solution to the wrong problem is useless, stating the problem is as important as building the solution. Hence the centrality of requirements engineering — also known as business analysis — in information technology.

Good requirements are among the most treasurable assets of a project. Bad requirements hamper it at best and doom it at worst.

In software development as practiced today, requirements are more often bad than good. What passes for requirements in too many projects is a loose collection of "use cases" or "user stories", revealing the kind of amateurish process that used to plague other tasks of software engineering such as design, programming and testing. While these solution-side tasks have benefited from enormous progress in the last decades, on the problem side requirements remain the sick part of software engineering.

The goal of this book is to redress the balance so that the requirements you produce will support rather than hinder your projects. It is not a theoretical treatise but a Handbook, devised to provide you with concrete and immediately applicable guidance.

THE MATERIAL

You will find in the following chapters:

- 1: A precise definition of requirements **concepts**, and a classification of requirement kinds.
- 2: A discussion of general requirements **principles**.
- 3: A **Standard Plan** applicable to the requirements of any project.
- 4: A review of the **quality** attributes for requirements and associated **verification** criteria.
- 5: Precise guidelines on how to **write** effective requirements.
- 6: A description of how to obtain requirements, a process known as **elicitation**.
- 7: A discussion of **use cases** and other scenario-based requirements techniques.
- 8: A presentation of the **object-oriented** approach to requirements.
- 9: An introduction to **formal** requirements, using mathematical rigor for precision.
- 10: An important kind of formal specification, **abstract data types**.
- 11: What it means for requirements to be "**complete**", and how to achieve this goal.
- 12: How to make requirements a core part of the project **lifecycle**.

As befits a practical and compact Handbook, the discussion focuses on concepts and uses only short examples for illustration. A Companion Book, *Effective Requirements: A Complete Example*, develops the requirements of an entire industrial case study from start to end, using the concepts of this Handbook and the plan of chapter 3.

OBSTACLES TO QUALITY

Why has requirements quality continued to lag while other aspects of software engineering have advanced? Lack of attention is not the reason. There are thousands of articles on requirements engineering, conferences that have been running regularly for decades, specialized journals, and several good books (you will find references to them in the Bibliographical notes and further reading section at the end of this Preface). Their effect on how industry practices requirements is, however, limited.

One of the obstacles has already been noted: the belief, in much of the software world, that doing requirements means writing a few scenarios of user interaction with the system: "use cases" or "user stories". While helpful, such a collection of examples cannot suffice. If used as a substitute for requirements, it leads to systems that do not perform well outside of the chosen cases and are hard to adapt to new ones. The industry needs to wean itself from use cases and user stories as the basis for requirements, and start viewing them in their proper role: as tools for the *verification* and *illustration* of proper requirements, produced by more appropriate techniques.

Another impediment is the widespread distrust of "upfront" activities — specifically, upfront requirements and design— sown by proponents of agile methods such as Scrum. Along with the undeniable improvements it has brought to the industry's practice of software construction, the spread of agile ideas has led many people to believe that requirements as separate software engineering artifacts are a thing of the past, and that you can just rush into coding, writing user stories as you go. In reality, *some* upfront work is essential: in no serious engineering endeavor can engineers proceed directly to construction without a preliminary phase of analysis and planning. Good software practices include requirements, whether you write them before or during development. In fact, as you will learn (see "Requirements Elaboration Principle", page 25, and the lifecycle discussion of chapter 12), you should do *both*. The principles in this Handbook are equally applicable to agile and more traditional ("Waterfall") projects.

DESCRIPTIVE AND PRESCRIPTIVE

We may expect anyone discussing a branch of science or engineering to start by precisely defining the objects of study. Unfortunately, the requirements literature lacks such meaningful and systematically applicable definitions.

It often compounds the problem by failing to separate *descriptive* and *prescriptive* elements. To study any discipline, you need to learn the basic notions involved before you learn right and wrong ways of doing things. Speed is distance traveled per unit of time; only after giving this definition can you start prescribing speed limits.

In software engineering and particularly requirements engineering, the standard sources have not reached that level of maturity. They are as long on advice — not always buttressed by objective justifications — as they are short on usable technical information, and many an author seems to find it natural to claim a role of director of conscience for stranded souls. Consider this definition of "requirement" from the IEEE standard on systems engineering:

> ***Requirement****: A statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability (by consumers or internal quality assurance guidelines).*

Although you would not guess it from its mystifying grammar (how does one parse "*product or process operational, functional, or design characteristic*"?), this definition is the result of years of work by an IEEE committee; numerous articles and textbooks cite it reverently. But it misses its purpose of defining the concept of requirements: it is instead trying to tell us what requirements *should* be (unambiguous, testable, measurable). Hold the preaching, please; first tell us what requirements *are*.

In its attempt at prescription, the definition is lame anyway: requirements quality involves much more than the criteria listed. In chapter 4 of this Handbook, devoted to defining requirements quality, you will find a set of fourteen quality factors. It is not possible to do justice to such a complex matter in the few lines of a definition. But consider the damage that this botched attempt at prescription does to the *description* (which should be the goal of a definition in a standard). If we only accepted "*unambiguous*" requirements as requirements, we would exclude many — probably most — requirements documents produced in practice. (Imagine a definition of "novel" specifying that the story must be absorbing, the characters compelling, the dialog sharp and the style impeccable. Bookstores would have to remove many titles from their "novel" shelves.) Requirements as we write them are human products; *of course* they will contain occasional ambiguities and other deficiencies! Not every one of their elements will be "testable or measurable". Perfect or not, however, they are still requirements.

Such confusion of the descriptive and the prescriptive is pervasive in today's standards. It mars what should have been the definitive standard on requirements (but ends up being pretty useless): the 2018 International Standard Organization's "Systems and software engineering — Life cycle processes — Requirements engineering", which you can purchase for some $300 to get such definitions as the following for "***requirements elicitation***":

> *Use of systematic techniques, such as prototyping and structured surveys, to proactively identify and document* ***customer*** *and* ***end user*** *needs*

(The underlined terms refer to other entries in the standard.) Requirements elicitation, covered in chapter 6 of this Handbook, is the process of gathering requirements from stakeholders. The cited definition only lists "*customers*" and "*end users*" as the source of needs, an obsolete view: it should refer to the more general notion of stakeholder (for which the standard actually has an entry!). The previous example used similarly imprecise and inadequate terminology by referring to acceptability by "*consumers*".

Even worse in the last entry is its failure to separate the definition of "elicitation" from the prescriptive fashions of the moment. Some committee member must have had a particular ax to grind: that *prototyping* is the best way to elicit requirements. (On prototyping for requirements, see 6.11, page 122 in this Handbook.) Another was pushing the idea of "*structured surveys*". They both got their two cents in, but at the expense of other widely used elicitation techniques (why leave out *stakeholder interviews* and *stakeholder workshops*, widely-used elicitation techniques discussed in chapter 6?). The result is a mishmash of partial prescriptions, not a usable definition.

The present text has a fair amount of advice, as one may expect from a Handbook. But it always defines the concepts first, and keeps the two aspects, descriptive and prescriptive, distinct. A prescriptive part, whether an entire chapter or just one section or paragraph, is marked at its start with the "Prescription" road sign shown here.

The first two chapters highlight the distinction: chapter 1 reviews and precisely defines the fundamental concepts of requirements; it is almost fully descriptive. Chapter 2 introduces general principles of requirements analysis and is almost fully prescriptive.

A BALANCED VIEW

One of the obstacles facing any serious discussion of software requirements is the dominance of two extremist schools with little tolerance for each other:

- "*Heavy artillery*": the more dogmatic fringe of the Waterfall, big-software-project school, which treats requirements as a step of the software lifecycle and insists that the subsequent steps cannot proceed until every single requirement has been spelled out.
- "*Guerrilla warfare*": the more dogmatic fringe of the agile school, which is suspicious of any "big upfront" activity (including upfront requirements and upfront design), and limits requirements to "user stories" (7.2, page 132), covering small units of functionality and written on-the-fly, interspersed with implementation.

Both extremes are unreasonable (and not endorsed by the wiser members of both schools). This Handbook takes a pragmatic stance on the place of requirements in the overall software devel-

opment process. Two of the "key ideas" summarized in the next section, "*Just Enough Requirements*" and "*Upfront **and** evolving*", reflect this flexible approach, which accommodates:

- Heavy-requirements processes, as may be justified for example in life-critical systems or others subject to strict regulatory processes.
- Light-requirements processes, as in web interface design or DevOps (12.4.3) projects.
- Anything in-between.

Each project is entitled to define the dosage of "a priori" and "as we go" requirements that best suits its context. This Handbook will, it is hoped, provide guidance and support in all cases.

KEY IDEAS

Successful requirements engineering demands a coherent approach with clear guiding principles. Here is a preview of core ideas that this Handbook will help you master and apply.

A Standard Plan. Requirements in industry, when just using an ad hoc structure, often follow the model plan of a 1998 IEEE standard. While good for its time, it has long outlived its relevance; we understand far more about requirements, and today's projects are vastly more sophisticated, calling for a more sophisticated plan. The plan presented in chapter 3 consists of four "books" covering the *four PEGS of requirements engineering* (Project, Environment, Goals and System), with a chapter structure covering all important aspects. It has been tried on a number of examples and fine-tuned over several years, with the goal of becoming the new standard.

A proper scope for requirements. Requirements are too often misconstrued as "the definition of the functions of the system". Such a view restricts the usefulness of a requirements effort. This Handbook restores the balance by covering all four PEGS of the requirements plan. All are equally important. "*Project*" covers features of the actual development project, such as tasks, resources and deadlines. "*Environment*" covers properties with which the development must contend, but which are not under its control because they come from physical laws, engineering constraints or business rules. "*Goals*" covers the business benefits expected from the project and system. "*System*" covers the behavior and performance of the system to be built.

Requirements as a question-and-answer device. The maximalist view of an all-encompassing requirements document, which must specify everything there is to know about a system (and in traditional "Waterfall" approaches, specify it ahead of any design or implementation), is in most cases expensive, unfeasible (as not all system properties *can* be determined early on), and over-reaching (as the project may not *need* to determine all of them early). Pushing this view on a project may lead to an equally damaging over-reaction from the team: a blanket dismissal of the importance of requirements. More productive and practical is a view of requirements as a technique for identifying key *questions* to be *asked* about the system, and *answering* these questions independently of design and implementation. This Handbook focuses throughout on this role of requirements as a question-asking and question-answering tool.

Not just documents. We will be less concerned with *requirements documents* in the traditional sense than with *requirements*. Elements of requirements appear not only in dedicated documents but in a variety of expected and unexpected places, from PowerPoint slide decks to emails. It is more productive to think of a repository (a database) of requirements, from which one can produce requirements documents if desired. The four books of the Standard Plan collect all necessary elements, across all four dimensions, but do not have to be written linearly.

Just enough requirements. Requirements are the focus of this Handbook, but they should not be the focus of software development. What counts is the quality of the systems you will produce. To reach this goal, you need to pay enough attention to requirements, but not so much as to detract from other tasks. This Handbook teaches how to devote to requirements the requisite effort — no less, and no more.

Upfront *and* evolving. The Waterfall-style extreme of requirements all done up-front then frozen, and the agile extreme of requirements (user stories) produced piecewise while you implement system components, are equally absurd. It is as irresponsible to jump into a project without first stating the requirements as it is illusory to expect this statement to remain untouched. The proper approach is to start with a first version (carefully prepared but making no claim of perfection or completeness) and continue extending and revising it throughout the project. This combination of up-front work and constant update avoids the futile disputes between traditional and agile views; it retains the best of both.

Requirements are software. Requirements are a software engineering product of the first importance, along with other artifacts such as code, designs and tests. They share many of their properties and can benefit from many of the same techniques and tools.

Requirements as living assets. As one of the fundamental properties they share with other software artifacts, requirements will inevitably undergo *change*. Correspondingly, they can benefit from *configuration management* techniques and tools for recording individual elements, their relations with others, and their evolution throughout the development process.

Taking advantage of the object-oriented method. The object-oriented style of decomposition structures specifications (of programs but also of systems of any kind) into units based on types of objects, rather than functions; then each function is attached to the relevant object type and the types themselves are organized into inheritance structures. This style has proved its value in the software development space, by yielding simple and clear architectures, facilitating change and supporting reuse. While it has long been known that the same ideas can also help requirements, they should be more widely applied in that space. This Handbook shows how to benefit from an OO style for requirements.

Taking advantage of formal approaches. Some parts of requirements demand precision, at a level that can only be achieved through the use of mathematical methods and notations, also known as formal. For most projects, the bulk of the requirements is informal — using a combination of English or other natural language, figures, tables… — but it is important to be able to switch to mathematics for aspects that have to be specified rigorously, for example if

misunderstandings or ambiguities could cause the system to malfunction, with potentially grave consequences. This Handbook shows that formal approaches are not an esoteric academic pursuit but a practical tool for requirements engineering, and explains how to benefit from them in a realistic project setting.

GEEK AND NON-GEEK

The charm as well as the challenge of requirements engineering is that it straddles geek and non-geek territory. Requirements describe how a software project and the system it produces interact with their physical and business environment (non-geek), but must do so with enough rigor and precision to serve as a blueprint for development, verification and maintenance (geek).

The geek/non-geek duality is apparent in the existence of two competing terms: what some branches of the Information Technology (IT) industry call "requirements engineering" is known in others as "business analysis". While nuances exist between these names ("Requirements engineering, business analysis", 1.2.5, page 6), for the most part they express a difference of focus: engineering versus business.

This Handbook does not take sides. It is intended both for IT professionals ("geeks") and for non-IT stakeholders ("non-geeks") wanting to understand how to make projects meet their needs. It ignores industry borders and applies to projects in both the engineering and business worlds.

THE AUTHOR'S EXPERIENCES BEHIND THIS HANDBOOK

A technical book is usually one of: practical advice, by a consultant; course textbook, by an academic; research monograph, also by an academic; prescription of standard practices, often by a committee. This Handbook does not fall into just one of these categories, but has features from each. It benefits from the author's experience across several professional roles.

Part of this background is the author's practice as a **software project team leader**. A successful project must avoid two opposite dangers: unprepared coding (jumping too early to implementation, without taking the time to define requirements); and "*analysis paralysis*", whereby you become so bogged down specifying requirements down to the last detail that you have no time left to implement them properly. Experience teaches how much effort to devote to requirements so that they guide and protect the development without detracting from it.

Another experience — **helping projects while they are under development** — confirms what many published studies have shown: that some of the worst deficiencies in software systems come from insufficient work on requirements (rather than mistakes in the design and implementation of the software). It is amazing in particular to see how a distorted invocation of agile ideas can damage a project: "*We are agile! We don't do any requirements! We just start implementing and add user stories as we go!*". A sure way to disaster. Agile methods — often used in a misunderstood form — serve here as a convenient excuse for sloppiness and laziness. In agile and less agile projects a consultant can help a development team produce a much better

system by prompting them, both upfront and throughout the development, to identify relevant stakeholders and devote the proper effort to requirements. This Handbook explains how to combine a significant but limited upfront requirements effort with a constant update and extension of the requirements throughout the rest of the development process.

Also part of the author's background for this Handbook is work as **software expert in legal cases**. Company C (customer) contracts out to company D (developer) to build an IT system. Things go sour and two years later they find themselves in court. C blames D for failing to deliver a working system, D blames C for failing to provide enough information and support. In comes a software expert, asked by the court to assess the technical merits. Sifting though tens of thousands of emails, meeting minutes, PowerPoint presentations, use cases, test reports and other project documents reveals major requirements-related problems. Sometimes they are the cause of the failure, sometimes just one factor, but they are *always* part of the picture. The flaws can be managerial (requirements did not receive enough attention); technical (requirements were not of good enough quality); human (D did not provide the right *business analysts*, C did not provide the right *Subject-Matter Experts* — see "Who produces requirements?", 1.10.2, page 16). In all cases, the expert's sentiment — kept to himself, since it's too late — is that the parties would have been better off devoting proper attention to requirements while the project was alive; and if they had to call on an expert, it would have been better to do so upfront (in the role of a project advisor, discussed in the previous paragraph) to secure the project's success, rather than now to help decide who pays and who receives millions in damages.

This Handbook benefits from numerous one-day or two-day **courses for industry** on requirements engineering and related topics, taught by the author to industry practitioners.

Also on the teaching side, the text relies on the author's **university courses** at ETH Zurich, Politecnico di Milano, Innopolis University and the Schaffhausen Institute of Technology on requirements engineering and more general software engineering topics (including agile methods). Such courses often include a development project with a requirements component. A particularly interesting experience was the *Distributed Software Engineering Laboratory*, taught for over a decade at ETH, and covering the challenges of software projects developed collaboratively across different sites. A key part of the course was a project conducted with several other universities and resulting in the full implementation of a system by student groups. Each group consisted of three teams located in different universities from different countries, with two or three students in each team. There is hardly a better way for students to realize the importance of requirements than when you have to interface your own part of the system with another written by people a few time zones away, from a different culture, and whom you have never met. Many students who took part in this experience have commented on how well it prepared them for the reality of distributed development (before Covid-19 made this setup even more prevalent), and how it helped educate them in fruitful requirements techniques.

BIBLIOGRAPHICAL NOTES AND FURTHER READING

The "Companion Book" mentioned on page viii is *Effective Requirements: A Complete Example* [Bruel et al. 2022].

Examples of the existing "good books" on requirements (page viii) include, on the practical side, [Wiegers-Beatty 2013], rich with examples from the author's practice as a consultant. On the more academic side, an important contribution is [Van Lamsweerde 2008] which covers the field extensively, focusing on goal-oriented requirements techniques; see also a textbook, [Laplante 2018]. Another requirements text is [Kotonya-Sommerville 1998]. [Pfleeger-Atlee 2009] is a general textbook on software engineering, but its almost 80-page chapter on requirements provides a good survey of the topic. Another software engineering textbook, older but still applicable, is [Ghezzi et al. 2002]. A classic text on software project management, [Brooks 1975-1995], includes some oft-quoted lines about the importance of requirements. An important source is the work of Michael Jackson and Pamela Zave, starting with an influential early paper, [Zave-Jackson 1997] and continuing with Jackson's own requirements books: [Jackson 1995] and [Jackson 2000]; a more recent compendium of the work of their school is [Nuseibeh-Zave 2010]. [Lutz 1993] is a classic study of software errors due to poor requirements.

The standards cited on page ix are the IEEE systems engineering process standard [IEEE 2005], and the ISO-IEC-IEEE requirements engineering standard [ISO 2018]. Another IEEE-originated standard is SWEBOK [IEEE 2014], the Guide to the Software Engineering Body of Knowledge. It still shows signs of immaturity (with such examples as "*a process requirement is essentially a constraint on the development of the software*", where "*essentially*", inappropriate in a definition, can only confuse the reader). It has, however, become more precise and rigorous over its successive editions (the latest one, referenced here, is the third) and serves as a good summary of accepted concepts of software engineering including requirements, the topic of its first chapter.

The Distributed Software Engineering Laboratory at ETH Zurich and elsewhere, initially called DOSE (Distributed and Outsourced Software Engineering), included a project developed collaboratively by students from different universities around the world, in which requirements played a key role. It led to numerous publications accessible from [DOSE 2007-2015].

ACKNOWLEDGMENTS

Special thanks are due the Schaffhausen Institute of Technology (sit.org) for providing an excellent environment for teaching and research. SIT is an ambitious new university destined to make a big splash in the technology world; this Handbook appears to be the first book produced by an SIT member since SIT's creation in 2019. It is important to express the key roles of Serguei Beloussov, the founder of SIT and definer of its vision, Stanislav Protassov, one of SIT's leading lights, and faculty colleagues Mauro Pezzè and Manuel Oriol.

Part of the context that led to this Handbook is the collaborative work, going back several years, of an informal research group on requirements whose members are spread between the University of Toulouse (IRIT, Université Paul Sabatier), SIT, and previously Innopolis University. The present work is in debt to the members of this group for many stimulating discussions and particularly for helping with the initial version of the taxonomy of requirements ("Kinds of requirements element", 1.3, page 6). They are Profs. Jean-Michel Bruel, Sophie Ebersold and Manuel Mazzara as well as Alexandr Naumchev, Florian Galinier and Maria Naumcheva.

The author's expert-consulting work in legal cases, and the resulting insights mentioned above, greatly benefited from collaboration with Benoît d'Udekem from Analysis Group.

The courses cited in the previous section yielded thoughtful comments by attendees, lessons from course projects, and insights from co-lecturers, teaching assistants and colleagues including, at ETH, Peter Kolb, Martin Nordio, Julian Tschannen and Christian Estler; at Innopolis, Alexandr Naumchev and Mansur Khazeev; at Politecnico di Milano, faculty members Elisabetta Di Nitto and Carlo Ghezzi in many thought-provoking discussions. A seminar at UC Santa Barbara in 2020 at the invitation of Laura Dillon and two talks in 2021, one for ACM, organized by Will Tracz, the other for IBM, at the invitation of Grady Booch, provided opportunities to refine the ideas and their presentation.

The author has had the privilege of being exposed early on and over the years to the work of pioneers in requirements engineering, people who really defined the field, and even in some cases to interact directly with them. Without in the least implying agreement, it is important to acknowledge the influence of such star contributors (a few of them not strictly in requirements engineering but in kindred areas, for example agile methods and software lifecycle models) as Joanne Atlee, Kent Beck, Daniel Berry, Barry Boehm, Grady Booch, Mike Cohn, Alistair Cockburn, Anthony Finkelstein, Carlo Ghezzi, Tom Gilb, Martin Glinz, Michael and Daniel Jackson, Ivar Jacobson, Capers Jones, Cliff Jones, Jeff Kramer, Philippe Kruchten, Bashar Nuseibeh, David Parnas, Axel Van Lamsweerde, Karl Wiegers and Pamela Zave. A number of them are members of the IFIP (International Federation for Information Processing) Working Group 2.10 on Requirements; attendance at one of their meetings provided many insights, as did regular participation in meetings of another IFIP committee, WG2.3 on Programming Methodology.

The friendly and efficient support of Ralf Gerstner at Springer, now for the third book in a row, is a great privilege.

The ETH Zurich library helped in obtaining the text of older articles. Alistair Cockburn kindly authorized using material from his book on use cases, [Cockburn 2001], for an example appearing in chapters 7 and 8; Bettina Bair kindly authorized reproducing her sample requirements document, devised for a course, [Bair 2005].

Comments received on early drafts of the text, particularly by from Mike Cohn, Lutz Eicke, Philippe Kruchten, Ivar Jacobson and Karl Wiegers, led to corrections and improvements.

Marco Piccioni provided support, comments and material over many years, and suggested exercises.The text immensely benefited from Raphaël Meyer's punctilious proofing. However much one would like to hope that no mistakes remain, chances are slim; the Handbook site referenced below will list corrections to errors reported after publication.

September 2022 (corrected printing)

HANDBOOK PAGE

Further material associated with this Handbook, including course slides, document templates for the Standard Plan of chapter 3 and links to video lectures (MOOCs) on requirements, is available at

requirements.bertrandmeyer.com

CREDITS

Cover picture: from "*Émailleur à la Lampe, Perles Fausses*" (lampwork enameler, imitation pearls), a plate in Diderot's and d'Alembert's *Encyclopédie* (1751-1766), by kind permission of the ARTFL Project at the University of Chicago.

Pages 168 and 178: detail from *A Pic-Nic Party* by Thomas Cole, Brooklyn Museum, photo by Bill Hathom on Wikimedia at upload.wikimedia.org/wikipedia/commons/0/09/Thomas_Cole%27s_%22The_Pic-nic%22%2C_Brooklyn_Museum_IMG_3787.JPG. See museum page at www.brooklynmuseum.org/opencollection/objects/1356.

RUP diagram, page 214: adapted from Wikimedia picture at commons.wikimedia.org/wiki/File:Development-iterative.png.

Contents