5

How to write requirements

From: *Handbook of Requirements and Business Analysis*, by Bertrand Meyer, Springer, 2022.
See book page at requirements.bertrandmeyer.com.

Requirements engineering is an act of intermediation: you gather requirements from stakeholders and other sources and put them in a form useful to the developers. The next chapter will explain how to gather ("elicit") requirements; this one explains how to write them.

Section 5.1 discusses when this writing process will happen, and where its results will appear (reminding us that in most cases we should consider not just a single "requirements document" but a set of documents). As a warning of the difficulty of the task, section 5.2 introduces the "seven sins of the specifier": common mistakes in requirements writing. 5.3 discusses the nefarious influence of repetition, and how to avoid it. A related issue, addressed in 5.4, is how to separate explanatory text from binding specifications. Requirements are not all written in natural language; 5.5 discusses the role of other notations such as graphics and tables. The remaining sections present rules for natural-language requirements. 5.6 criticizes and improves a few poorly written examples. Switching from "how not to" to "how to", 5.7 presents the fundamental style rules, and 5.8 explains how to handle "TBD" (To Be Determined) elements. Section 5.9 addresses the specific needs of the Goals part of requirements. (The other sections apply to all four "books" of requirements in the plan of chapter 3: Project, Environment, Goals and System.) The last section covers a classic example, small but instructive, showing how much one can err by not following the principles expounded in the rest of the chapter.

As with other prescriptive parts of this Handbook, remember to avoid the risk of perfectionism ("Requirements quality", 2.5): quality of requirements writing is but one component of quality of the requirements process, which itself is but one step towards quality of the final system — the goal that really matters. This chapter is not about enforcing perfection. It simply presents professional standards for requirements writing, whose application costs much less than the damage that poorly written documents will cause.

5.1 When and where to write requirements

The two processes of gathering and writing down requirements do not need to happen (as sometimes suggested in older views of software engineering) as two successive steps. An iterative, interleaved scheme is usually more productive: gather some, write some, repeat.

Iteration provides a feedback loop. As you perform requirements gathering activities such as interviews and workshops, the very act of writing down what you are learning helps you identify forgotten questions, which you can try to fit into the next gathering step. The observation yields the first principle of this chapter:

Requirements Writing Principle

Start writing requirements as you gather them.

This advice is part of a general trend in modern software engineering — most vividly illustrated by the spread of agile methods — away from strictly sequential, Waterfall-like processes, towards more iterative ones. It is in line with the advice that the requirements process as a whole does not need to happen only at the beginning of the project, but should continue throughout development. (The more complete form of the simplified cycle given above would be: gather some, write some, *develop* some, repeat.)

Along with the "when" part, we must also ask *where* requirements will appear. Remember ("The form of requirements", 2.1.6, page 26) that we can seldom expect that all the wisdom about a complex modern project will fit in a single, linear "requirements document". Requirements information will appear in many different places, some of which are beyond the control of the project team and may in fact predate the project ("pre-requirements" as defined in 2.1.6).

5.2 THE SEVEN SINS OF THE SPECIFIER

Before getting to positive advice about requirements writing, we start with a list of deficiencies commonly found in requirements documents.

5.2.1 The Sins list

Some of the danger signals reviewed below are simply violations of desirable properties of requirements, discussed in the previous chapter and elsewhere in this Handbook. Grouping them into a separate list will help you remain alert to their occurrence when you encounter initial versions of requirements specifications, or are writing such specifications yourself. The table below lists these "Seven Sins of the Specifier", coming from a classic paper in the field. The following subsections provide further comments on some of the categories.

In the table's entries, remember that a "relevant property" is a feature of the Project, Environment, Goals or System that can affect stakeholders (the complete definition was in 1.2.3).

The Seven Sins of the Specifier

Deficiency	Definition
Noise	A requirements element that carries no information on any relevant property (5.2.2). Variants: *remorse* (5.2.3); *repetition* (5.3).
Silence	The existence of a relevant property not covered anywhere in the requirements (5.2.2).
Contradiction	Two or more requirements elements that define a relevant property in incompatible ways. Variant: *falsehood* (5.2.4).
Ambiguity	A requirements element from which a reader may understand a relevant property in different ways. Variants: *synonyms* (5.2.5); *etcetera list* (5.2.6).

Wishful thinking	A requirements element that defines a relevant system property in such a way that it is not possible to determine whether a candidate solution satisfies it.
Overspecification	A requirements element that does not correspond to a relevant property, but to features of a possible design or implementation (4.7.3, page 59). An insidious variant is *operational reasoning* (see 9.5.4).
Dangling reference	A mention, in a requirements element, of a property that should be defined elsewhere in the requirements but is not.

The list does not include everything that can go wrong in requirements; after all, we had fourteen quality factors in chapter 4, not just seven, and each of them can — bad news — be violated in several ways. We are focusing on requirements *writing*, not the full requirements process, and on particularly common mistakes which — here comes some good news — you can avoid if you are aware of them.

5.2.2 Noise and silence

Noise is frequent in requirements; so is silence. The reason is a human tendency to go for the easier task. Sometimes it is easy to add information that is not relevant but makes the specifier feel good, and tempting to forget aspects that are trickier to explain precisely. The blame for noise as well as silence can lie with either or both of:

- Stakeholders who babble about unimportant things and skip important ones. The remedy is to focus on stakeholders of highest value to the project ("Assessing stakeholders", 6.5, page 109) and keep asking them relevant questions ("Ask effective questions", 6.10.4, page 119).
- Requirements engineers or business analysts, who fail to ask some important questions. The remedy is to perform reviews and verification of requirements, in addition to the requirements writers' self-assessment, to check the requirements for relevance and completeness.

5.2.3 Remorse

Remorse is a form of noise, which arises when a requirements document discusses a certain component (typically, of the System), and further in the text mentions the possibility that it might not exist. We will see an example below (5.6.3): a specification of an error report produced by a program, and later on the note that if there is no error there shall be no such report. Reasonable enough, but coming too late! This practice, while common, causes confusion and uncertainty. The proper approach, to specify an optional component, is the logical one:

- Upon first introducing the component, mention that it is optional.
- At the same place, specify the conditions under which it will exist (for example, an error report shall be produced if and only if the processing produces at least one error, where the specification defines what exactly constitutes an error).
- Only then describe the properties of the optional component for the cases in which it exists.

5.2.4 Falsehood

Contradiction does not even need to include two statements at odds with each other. *One* statement can be wrong by itself. This kind of wrongness is not the same as the case of an incorrect requirement, in the sense of correctness, the very first of the quality factors (4.1). A requirement is incorrect if it does not reflect actual needs (for a goal, system or project requirement) or constraint (for an environment requirement). In requirements writing, a requirement is a falsehood if it includes a statement of a false property. We will see an example at the end of this chapter: a statement that a "line", in a text, is a sequence of characters (other than new-line) appearing between two new-line characters — plainly wrong since it does not cover the case of the first and last lines (which respectively have no preceding and ending new-lines).

The remedy against falsehoods is simple, although not miraculous: in addition to checking every piece of requirements against the actual intent (to ensure correctness) and against others (to avoid contradiction), make sure that it is internally (that is to say, just by itself) sound.

5.2.5 Synonyms

Ambiguity can arise from the use of multiple names for the same concept. As discussed in more detail below ("Repetition", 5.3, page 75), technical writing differs from literary writing in promoting rather than eschewing the use of a single term for a given concept. Referring to the same concept as "apparatus" in one paragraph and "device" in another may lead the reader to conclude that they denote different kinds of things. Hereby lies a principle:

No-Synonym Principle

In requirements writing, enforce a one-to-one correspondence between important concepts (of the project, environment, goals and system) and their names.

5.2.6 Etcetera lists

A frequent case of ambiguity is the "etcetera list": the enumeration of the variants of a certain notion, ending with "etc." or "and so on" or "and others". For example, "*the system shall provide user interfaces in English, French, Spanish, Mandarin etc.*" To system developers, this list is confusing: should the system support any other languages, and if so which ones?

Such a phrasing is never appropriate. Instead of it, use one of the following two solutions:

- Give the complete list if possible. A little more work for the requirements engineer, but saves headaches and mistakes for the developers.
- If the definitive list is not closed at requirements time and the variants listed are just examples, say so by using the phrasing "*such as*", "*including*" or "*for example*" and — importantly — indicating where the complete list will appear. In listing specific cases, make it clear whether they are just examples (you can then use "*such as…*") or a required subset ("*including, but not limited to…*"). There is a difference between

> *The system shall provide user interfaces in languages such as (for example only) English, French, Spanish and Mandarin. The precise list of languages to be supported appears in the regularly updated Language Support List at https://…*

and

> *The system shall provide user interfaces in several languages including at least the following: English, French, Spanish and Mandarin. The full list appears in the regularly updated Language Support List at https://…*

The first is non-committal as it only gives examples and refers the reader to a dynamically evolving document (which could also be a chapter of one of the requirements books, since per this Handbook's principles requirements are a living product). The second also has this dynamic component but specifies a fixed minimum.

Either of these styles may be better depending on the circumstances, but the "etcetera" style is not acceptable.

Note that inheritance, in object-oriented requirements (chapter 8), provides an elegant way to add new variants to a predefined notion.

5.3 REPETITION

One of the forms of noise (the first of the "seven sins" of 5.2) is a common plague of requirements which deserves particular attention: repetition.

Why is repeating relevant properties bad? The answer is that it causes multiple forms of damage:

- Giving the same information twice is a waste of space, one of the reasons why requirements documents can become large and unwieldy. Unreasonably long documents make requirements harder to comprehend and may even lead developers and others to start ignoring them and miss key aspects of the problem.
- In practice, repeated elements are often not *exactly* repeated. Identical copy-paste does happen (and is to be banned, since a *reference* to the source is always preferable, as will be explained below); but most of the time repeated information is repeated in a slightly different way. Without the requirements writers realizing it, the different phrasings may either lead various readers (particularly, various developers) to understand a property differently, or lead some of them to wonder whether there is an inconsistency. It is better to ensure that all requirements are distinct, and that each clearly defines a property and its variants.
- Finally, repetitions cause trouble when requirements evolve, as they will ("Requirements Evolution Principle", page 23). Even if you have been able to root out contradictions among the initial duplicates or quasi-duplicates, it is hard to avoid the emergence of new ones as these elements start evolving and diverging.

The following principle reflects this analysis:

No-Repetition Principle

In requirements writing, never specify a property more than once.

This rule is sometimes known as "Single Point of Maintenance". In slogan form:

Say it well; say it once.

The principle as stated applies to requirements documents. We have seen that requirements can integrate existing outside elements (such as emails and external documents). The project has no control over them; they may contain repetitions (and redundancies with internal documents). In the project's own documents, repetition is to be banned.

While the principle is strict and suffers no exception, it does not exclude some forms of redundancy which are not repetition in the proper sense and deserve special consideration:

- *Explanations* complementing the specification of a relevant property, although even this case should be handled restrictively (5.4 below).
- *Illustrations* of a property in a different notation, which are not repetition in the proper sense but do require care as well (5.3).
- *Refinement*: providing more detail on a property already specified in a general way. Refinement is not repetition but a presentation technique, avoiding the need to specify a complex property in just one chunk. For example the specification of wire transfers for a banking system might state "the rules in this section govern transfers to countries in Europe, as defined in paragraph Y", where paragraph Y states "*for the purpose of transfer rules in section X, 'Europe' shall denote the countries of the European Union as of 1 January 2022 plus Switzerland and* […]".

It is hard to find an excuse for true repetition since a suitable alternative is available: when you need to mention a relevant property (of the Project, Environment, Goals or System) that also appears somewhere else, you can choose one of the two places to state it, and in the other place *refer* to that statement. This observation leads to a companion of the above principle:

Reference Principle

Do not repeat: refer or refine.

Text processing systems help apply this principle. In the old days, populating a document with cross-references (as in "the *Seven Sins of the Specifier* appear on page 72 as part of section 5.2.1") was scary, because you knew that the slightest change would force you to update all references or risk releasing an error-ridden text. Today's tools automatically take care of the updating. Good requirements documents rely on this technology to make sure that every property is defined in one place, then used whenever needed through a reference to that definition.

A practical tip for such references:

Advice: formatting references to concept definitions

When using a requirements concept that has a precise definition elsewhere in requirements documents, use a consistent format, and (in an electronic version) make the reference a hyperlink to the definition.

A typical format convention is to underline such hyperlink references. For example (assuming this Handbook were a requirements document, which it is not):

Another way of stating that requirements documents should not include noise is to say that they should only specify relevant properties.

where the underlined part is a hyperlink to the definition of "relevant property" in 1.2.3, page 5. (Elsewhere, this Handbook marks hyperlinks with a different color rather than underlining.)

5.4 BINDING AND EXPLANATORY TEXT

The primary purpose of requirements is to specify properties of the Project, Environment, Goals and Systems. A committed requirements writer, intent on producing a useful document, will often feel an additional urge to *explain* the specification. Most of the time, this urge should be resisted.

"Most of the time" because a few specific parts of requirements do leave a role for explanations. In the Standard Plan of 3.1 they are the following chapters:

The only places where explanations are legitimate (in Standard Plan of 3.1)

Chapter	*Title*	*Why an explanation may be justified*
P.2	Imposed technical choices	It may be useful to recall the strategic decisions that stand behind some of these choices.
P.7	Requirements process and report	The description of how the requirements process will be conducted, and the report on how to proceed, may benefit from a more discursive style.
G.1 to G.3	Context & overall objective, current situation, expected benefits.	By their nature, these chapters have an explanatory spirit.
G.6	Limitations and exclusions	It may be useful to explain why some parts are not covered by the system.
S.5	Prioritization	Although mostly a specification, this chapter may spell out the reasons for some of the specified priority decisions.

Explanations appearing in these contexts often express "justifications", one of the legitimate kinds of requirement, introduced in 1.9.

Outside of these chapters, the focus of requirements documents should not be on explanation but on specification, descriptive and prescriptive. (Mostly descriptive for the Environment and Goals, mostly prescriptive for the Project and System.)

The temptation to explain comes from good intentions: requirements writers want to produce clear, helpful documents. The rationale for *not* yielding to this impulse is the same as for the ban on repetition (5.3): avoiding contradiction. Assume a specification, in an airline booking system, that only adult passengers may occupy seats on an exit row, and a supposedly helpful explanation that "*there is no guarantee that people 18 or younger will understand safety instructions*". In fact, the age limit defining an "adult" varies, and even if it is 18 it usually includes people of that exact age (whereas the phrasing "*18 or younger*" bars them — did you notice the discrepancy?). A programmer seeing the explanation might implement an incorrect rule.

The more you try to repeat things, the higher the likelihood that mistakes of that kind will creep in. The better approach is in most cases to make the specification clear and precise, so that it does not *need* an explanation. Specification beats explanation. Say it well; say it once.

For cases in which you feel the bare specification is too dry and does require explanations, you should limit the risk of confusion by clearly separating the two kinds of elements: those that specify, and those that explain. You will find a model for this approach in good *standards documents*. A standard is prescriptive: it sets parameters that an entire industry must follow. Often, the specification by itself is terse and hard to understand; explanations will be welcome. A common practice in standards is to mark explanatory text explicitly as such, using appropriate typographical conventions, so that only the non-explanatory parts have prescriptive value.

An example, extracted from the standard for the Eiffel programming language, appears on the next page. Note how the various official components are marked with their nature (definitions, syntax, validity), and how "informative text" uses a distinctive format. This is an excellent practice, recommended for all requirements, not just standards. It encourages requirements writers to reflect on what is truly binding on the developers and what is simply meant to help them. (A similar convention should apply to implementation hints, covered in 4.7.4, page 59.)

A note on the pragmatic side: of all the requirements writing advice in this Handbook, this one is among the least commonly applied in today's practice. But it is not particularly hard to enforce, and produces a significant benefit.

The following principle summarizes explanation-related advice:

Explanation Principle

E1 In requirements, favor specifications over explanations.

E2 When explanatory elements are indispensable, devote special care to checking that they do not contradict the specification elements, even in small details.

E3 In requirements texts (and other forms of expression of requirements), use a clear and explicit convention to ensure that any explanatory elements stand out as distinct from specification elements.

Levels of discourse in a programming language standard

8.9.18 Definition: Check-correct

An effective routine *r* is **check-correct** if, for every Check instruction *c* in *r*, any execution of *c* (as part of an execution of *r*) satisfies its Assertion.

8.9.19 Syntax: Variants

Variant ≜ **variant** [Tag_mark] Expression

8.9.20 Validity: Variant Expression rule **Validity code:** *VAVE*

A Variant is valid if and only if its variant expression is of type *INTEGER* or one of its sized variants.

8.9.21 Definition: Loop invariant and variant

The Assertion introduced by the Invariant clause of a loop is called its **loop invariant**. The Expression introduced by the Variant clause is called its **loop variant**.

8.9.22 Definition: Loop-correct

A routine is **loop-correct** if every loop it contains, with loop invariant *INV*, loop variant *VAR*, Initialization *INIT*, Exit condition *EXIT* and body (Compound part of the Loop_body) *BODY*, satisfies the following conditions:

1 {**True**} *INIT* {*INV*}

2 {**True**} *INIT* {*VAR* ≥ 0}

3 {*INV* **and then not** *EXIT*} *BODY* {*INV*}

4 {*INV* **and then not** *EXIT* **and then** (*VAR* = *v*)} *BODY* {0 ≤ *VAR* < *v*}

Informative text

Conditions 1 and 2 express that the initialization yields a state in which the invariant is satisfied and the variant is non-negative. Conditions 3 and 4 express that the body, when executed in a state where the invariant is satisfied but not the exit condition, will preserve the invariant and decrease the variant, while keeping it non-negative. (*v* is an auxiliary variable used to refer to the value of *VAR* before *BODY*'s execution.)

End

5.5 NOTATIONS FOR REQUIREMENTS

A requirement is (1.2.4, page 5) "*a relevant statement about a project, environment, goal or system*", where a "statement" is (1.2.2, page 4) "*a human-readable expression of a property*". As was discussed in the introduction of these concepts, the "expression" of the property may take place in various notations, including:

- Natural language, such as English.
- Graphical notations, using either precisely defined rules such as those of UML (Unified Modeling Language) or ad hoc conventions (as in informal diagrams of system structure).
- Tabular notations. Tables can be useful to specify behavior that depends on combinations of various parameters. A typical example appears below.
- "Formal" notations, such as mathematical specification languages, or programming languages such as Eiffel, used for description (rather than implementation) purposes.

The rest of this section takes a look at the specific demands of each of these forms, then discusses (in 5.5.5) how to combine them.

5.5.1 Natural language

Natural language accounts for the bulk of requirements, in both longer traditional versions ("requirements documents" in "Waterfall" style) and shorter versions (use cases, user stories).

The advantages and limitations of natural language are clear. On the positive side, it is the only notation that everyone understands without specific training. It is flexible and lends itself to expressing nuances; it can be used for explanation (5.4) as well as specification. On the negative side, natural-language statements often lack the precision required for specifying delicate properties of complex systems. Also, natural language can be discursive and wordy.

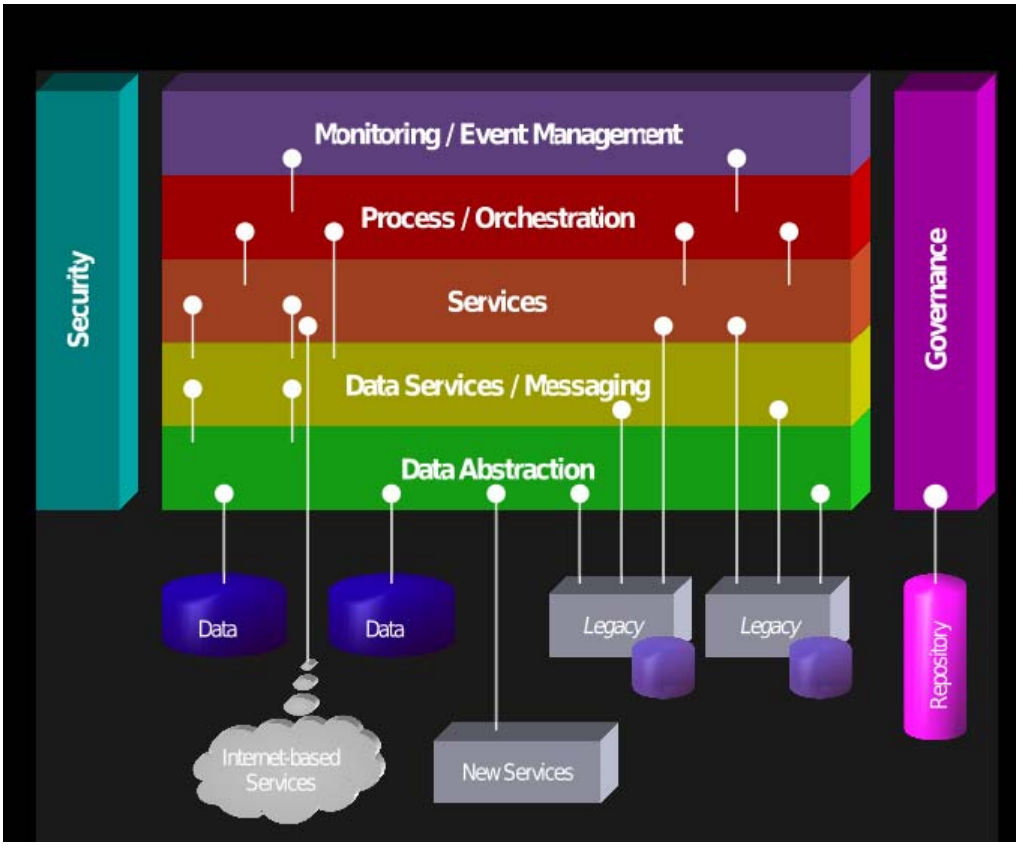
Where you need precision, you can complement natural language by formal specifications (5.5.3 below and chapter 9). For high-level overviews, graphical illustrations (5.5.2) can help.

Natural languages as applied to requirements are not quite the same as in their everyday use. Different rules apply: some (although not all) of what is considered good style in a novel can be bad for requirements, while a novel written in good requirements style would make for awful reading. Where literary pieces benefit from variety and surprise, requirements texts call for regularity; a good requirements document should be boring. A later section of this chapter introduces the specific rules: "Style rules for natural-language requirements", 5.7, page 88.

Going beyond this observation, some requirements methods use restricted versions of natural language, allowing only certain turns of phrase. See the bibliographic section for pointers.

5.5.2 Graphical notations

Graphical illustrations can be a great help for understanding. They are particularly appropriate in the description of Goals, to give a general idea of possibly complex problems and solutions.



SOA meta-model (Source: Wikipedia)

Using pictures in requirements calls for caution. Too often (with the cliché that "a picture is worth a thousand words" as implicit cover) technical texts flash fancy graphics without bothering to define the precise conventions being used. Such boxes-and-arrows diagrams are particularly common in networking, as in the richly colored figure on the preceding page, from Wikipedia, describing components and connections of a Service-Oriented Architecture.

This example typifies the problems found in many graphical illustrations:

- What exactly does it mean to have a particular layer (such as "Services") above another ("Data Services / Messaging")? What is the implied relationship between them?
- What is the meaning of the white connectors between layers (the ones that look like tadpoles caught in a snowstorm)? If there is a general rule that each layer relies on the one immediately below, why are there three connectors between "Process / Orchestration" and "Services" just below and just two between "Monitoring / Event Management" and "Process / Orchestration"? Why do some connectors go through intermediate layers, such as from "Process/Orchestration" directly to "Data Services / Messaging" and from "Services" to "Legacy"? Are these cases violations of traditional information hiding rules?
- Why do some elements, such as "Legacy", appear twice? Is it a convention to suggest elements that may appear in two or more instances, such as "legacy" elements and (at the same level) "data"? But are we then to understand that there is only one of "New Services" and of "Internet-based services"?
- The layers are of different nature. The "Data Services / Messaging" layer offers APIs (program interfaces) for certain network services. "Data Abstraction", on the other hand, is just a methodological concept (see chapter 8). Layering apples on top of oranges is confusing.
- The role of the vertical boxes on the sides is unclear. They seem to suggest that Security and Governance apply across all levels (as "cross-cutting" concerns). Since it is hard to think of an endeavor of any kind in information technology — or, for that matter, any other field — in which this observation does not apply, it seems hardly essential to the specification of Service-Oriented Architecture. Assuming the boxes do merit their spots in the figure, their placement is puzzling: does Security cease to be a concern for the lower levels, such as "New Services", to which the box does not extend? Why is there a tadpole from "Governance", a strategic concern, to "repository", a technical artifact? (Presumably having a repository helps governance, but the repository will also help "Process/Orchestration", "Monitoring/Event Management" and just about all other horizontal layers. Puzzling.)

Pictures of that kind, as strong on impressiveness as they are weak on precise meaning, are common in technical documentation.They are prized because they give the reader a quick grasp of the structure of a system. This impression can, however, be deceptive.

If you use such pictures, you should pay particular attention to the **connectors** between elements of a diagram — typically, lines or arrows of some kind. Too often, diagram authors include an arrow to indicate that component A somehow relies on component B, or interacts with ("talks to") B in some way, or knows about B, or uses B, or is just related to B in some unspecified sense. Such vagueness is inappropriate in requirements. **Beware of boxes bearing arrows**.

Pictures can do more harm than good unless they explicitly state the meaning (also known as *semantics*) of all symbols. As an example (although not from a requirements document), the figure on page 139 at the beginning of section 8.3 of this Handbook uses two kinds of arrow (for the "client" and "inheritance" relations in object-oriented programming) and an asterisk symbol; their semantics is specified in the figure's caption.

Along with arrows connecting components, the symbols representing the **components** themselves need to have a precisely specified semantics. If you are variously using squares, rectangles, ellipses, circles and other shapes, make sure they correspond not to your whim but to distinct concepts, each explicitly stated.

If a set of a common conventions apply to several figures, you should — in line with the No-Repetition Principle — specify them in a single place (instead of repeating them in separate captions), and refer to that place in each figure.

In the exercise of making sure that each graphical symbol corresponds to a specific meaning, you will often be led to simplifying the figures. Colors are a typical example: after some initial exuberance, supported by the power of modern graphical design tools, you may realize that you are using too many colors without justification. A color change, like any other graphical convention, should represent a change of concept. Otherwise, go for the drab! Divert your artistic creativity to other pursuits; your requirements pictures may at the result be less exciting but they will also be less confusing and more useful to the people who really need them.

Summarizing graphics-related advice:

Picture Principle

Graphical illustrations in requirements texts must only use symbols — in particular those representing components and their connectors — with a precise semantics, defined in the illustrations themselves or in a common reference.

Every pictorial variation must reflect a semantic variation.

A complementary piece of advice is to ensure that there is no contradiction between pictures and texts (and other notations that may appear in the requirements). It is developed further in "Combining notations", 5.5.5, page 84 below.

5.5.3 Formal notations

For specifications requiring precision, human civilization has devised a solution: mathematical notation. "$\forall x: \mathbb{R} \mid cos^2\ (x) + sin^2\ (x) = 1$" is vastly superior to all other ways of expressing the same property (including the natural-language statement that "*the sum of the squares of the cosine and sine functions applied to any real number, the same for both, is always equal to one*").

Mathematics-like notations, known as *formal* notations, and the associated *formal methods* have gained ground for software requirements. They do not have to replace natural-language requirements entirely, but can complement them for parts that require precision. The important topic of formal specification occupies the entirety of chapter 9.

5.5.4 Tabular notations

Tabular notation is a variety of formal specification using a simple spatial representation, easily understood even by people who may be resistant to the use of usual mathematical formalisms.

Consider a specification that expresses the value of a certain property as resulting from one or (more interestingly) two other properties. In the Open Systems Interconnection (OSI) model, for example, various networking features such as error recovery and the ability to reinitiate a connection are available, or not, depending on the class of transport protocols, of which there are five, TP0 to TP4. The property of interest is "feature available or not?"; the two properties that condition it are the networking feature and the transport class.

Mathematically, this specification is a case of a well-known notion: a boolean-valued function *present* taking two arguments. We can state that *present* (*Error_recovery*, *TP0*) has value false, *present* (*Error_recovery*, *TP1*) has value true and so on. But a tabular representation — as used in the OSI Wikipedia page — also does the job, in a concise and readable form:

Feature name	TP0	TP1	TP2	TP3	TP4
Connection-oriented network	Yes	Yes	Yes	Yes	Yes
Connectionless network	No	No	No	No	Yes
Concatenation and separation	No	Yes	Yes	Yes	Yes
Segmentation and reassembly	Yes	Yes	Yes	Yes	Yes
Error recovery	No	Yes	Yes	Yes	Yes
Reinitiate connection[a]	No	Yes	No	Yes	No
Multiplexing / demultiplexing over single virtual circuit	No	No	Yes	Yes	Yes
Explicit flow control	No	No	Yes	Yes	Yes
Retransmission on timeout	No	No	No	No	Yes
Reliable transport service	No	Yes	No	Yes	Yes

[a] If an excessive number of PDUs are unacknowledged.

Classes of Transport Protocol (Source: Wikipedia)

Tables also work well for specifying functions of one argument (rather than two), as in the example (about the simplest possible one) of the negation function in logic:

a	**not** *a*
True	**False**
False	**True**

Tables often beat all other notations when you need, as in these examples, to specify the precise value of a function of one or two parameters, each of which takes one of a fixed set of values.

In addition to being readily supported by spreadsheet tools such as Excel, tables benefit from being easy to change as the understanding of the requirements advances or the requirements themselves change. (We may note here the connection with the software design and programming technique of "table-driven computation", which computes results not through a specific algorithm, hard-coded into the program, but by looking up a table. The table can be represented as data, easier to change than program text. In this case the concept of table is applicable to more than two dimensions.)

For requirements, tables lose their appeal for more than two parameters — except if one of the parameters only has a small number of possible values, in which case you can use several tables — and for parameters that take too many possible values, or have a continuous (rather than discrete) range. Within their scope of applicability, however, they can be an excellent requirements specification technique, more precise than natural language and more widely understandable than other mathematical notations.

5.5.5 Combining notations

The four kinds of notation just reviewed — natural language, pictures, mathematics and tables — are the principal ones available for specifying requirements. All have many variants (reviewed in a survey article cited in the bibliographical section). They are not exclusive; in particular, while natural language in some form remains dominant in the industry's practice, it benefits from complements in other notations, for example graphics for illustration and formal or table notations for elements that require a precise, rigorous specification.

The caveat here follows from the discussions of repetition and explanation (5.3 and 5.4). The risk to keep in mind is that requirements elements expressed in different notations lead to understanding a certain property in slightly different ways. Hence the rule, which extends the No-repetition and Explanation principles to the case of multiple notations:

Multiple Notation Principle

In requirements using more than one notation, make sure that the binding specification of every property unambiguously appears in only one of notations.

If elements in more than notation pertain to a common property, make sure that the requirements clearly indicate which one is binding, and label the others as explanatory.

(Again: *say it well, say it once*.) The binding specification could be in natural-language text, with a figure illustrating the concepts. In this case, the figure caption should state "This figure for explanatory purposes only; see specification in section x.y", or equivalent. Conversely, a figure or table could serve as a precise specification (like, in this Handbook, the diagram of possible references between books on page 37), and a text could comment on it; such text should be labeled as "informative", per the conventions illustrated in the standard extract of page 79.

Work on *multirequirements* (see the bibliographical section) goes further than the last principle by proposing a requirements methodology that interleaves descriptions at three levels: natural-language texts; formal descriptions in a high-level language; graphical representation.

5.6 SOME EXAMPLES: BAD, LESS BAD, GOOD

In the spirit of grammar and style textbooks ("*do not write: …, write instead: …*"), here are some examples of requirements writing that is subject to improvement. Most come from various authors' publications, listed in the bibliography section.

5.6.1 "Provide status messages"

An example cited by Wiegers and Beatty: "*The Background Task Manager shall provide status messages at regular intervals not less than 60 seconds.*" Objections:

- Lower bound, but no upper bound! Is an interval of 5 minutes OK? An interval of two hours? Of a year?
- "*Provide status messages*". Is it OK to flash the messages for a couple of seconds and let them go away (like Skype alerts on a desktop)? Probably not. We need to specify whether, how and how long the messages should stay.

Wiegers and Beatty's proposed replacement:

> *The Background Task Manager (BTM) shall display status messages in a designated area of the user interface:*
> - *The messages shall be updated every 60 plus or minus 10 seconds after background task processing begins.*
> - *The messages shall remain visible continuously.*
> - *Whenever communication with the background task process is possible, the BTM shall display the percent completed of the background task.*

This rewrite has some improvements:

- It corrects the two deficiencies listed above.
- It indicates that the user interface design must provide a place for the error messages, without overspecifying the form of that UI element.
- It identifies the mechanism being specified under a precise name (an acronym, "BTM", which should have an entry in the glossary, see 6.4).

It is not, however, itself immune to criticism:

- It is much longer. Is the four-fold increase in the number of words truly justified? Wordiness leads to huge, hard to manage and eventually ignored requirements documents.
- One mention is either incorrect or infeasible (remember "Correct", 4.1 and "Feasible", 4.6): since screen space is finite and fonts must be large enough for readability, one cannot keep all messages "*visible continuously*". It should probably say "*accessible*" continuously (for example with a scroll list, although as noted there is no need to prescribe a particular UI style).
- The last part (about communication) is a new requirement, not deducible from the original. Maybe it is justified, but it also can raise a suspicion of overspecification.

Exercise 5-E.1, page 102 asks you to provide a replacement that addresses these problems.

5.6.2 The flashing editor

Another example cited and criticized by Wiegers and Beatty: "*The XML Editor shall switch between displaying and hiding non-printing characters instantaneously.*" Criticism:

- When does it "switch"? Does it decide on its own accord one minute to display non-printing characters in a file, such as backspace, and the next minute to hide it? Probably not (editors are not expected to possess free will), but the trigger should be stated.
- "Instantaneously". What is instantaneous to me may seem like an eternity to you. Such words have no place in a proper requirements text.

Wiegers and Beatty's replacement, correcting these deficiencies:

> *The user shall be able to toggle between displaying and hiding all XML tags in the document being edited with the activation of a specific triggering mechanism. The display shall change in 0.1 second or less.*

Indeed better. At the risk of nitpicking, we note that there is no obvious need for both the verbs "toggle" and "trigger", and that "*with the activation of a specific triggering mechanism*" is noise: by definition, any operation that an editor provides to its user is available through "a specific triggering mechanism" — what else could one expect? Since everything is clear and the replacement remains fairly concise, we would probably leave it alone in an actual requirements document, but for the sake of perfectionism (in a deviation from the Requirements Effort Principle), exercise 5-E.1 asks you to improve the text further if you can.

5.6.3 Always an error report?

One more Wiegers-Beatty example: "*The XML parser shall produce a markup error report that allows quick resolution of errors when used by XML novices.*" Its critique is left to the reader. The authors' proposed replacement is:

> *After the XML Parser has completely parsed a file, it shall produce an error report that contains the line number and text of any XML errors found in the parsed file and a description of each error found.*
>
> *If no parsing errors are found, the parser shall not produce an error report.*

This correction removes the ambiguous "*when used by XML novices*", a vague, subjective criterion no better than "*instantaneously*" in the preceding example. Who is an XML novice? How do we know what it takes for error messages to enable such a user to correct the error? (We do not.) More modestly but more usefully, the revised version says what basic information the error message should provide. This replacement is still subject to discussion:

- It has opted for an error report produced only after full parsing. This property was not in the original. It may be overspecification, since it excludes the possibility of reporting errors on-the-fly. Why not leave this aspect open, as a user interface choice to be made later? What matters is that the parser will be able to process input containing syntax errors and detect those errors, not *when* it will report them.
- The new text correctly indicates (unlike the original) that the system should only produce an error report if there is an error. This property may sound obvious, but the phrasing of the original version implied that there would always be a report, which is probably incorrect.
- The way the new version adds this property, is, however, a case of the "**remorse**" syndrome in requirements documents. Remorse is the phenomenon of describing some element (of the project, environment, goals or system) as if its existence was a done deal, only later suddenly to state that it might not exist after all, leaving the reader confused. Technically, remorse is a case of *contradiction*: in the first paragraph we are told that the processing "shall produce" — strong words! — an error report, and in the second that maybe it will not. The damage is minor since the two statements are close to each other, but in a complex requirements specification they might lie further apart, or even in different documents, and cause misunderstandings.

Again, can you do better? (Exercise 5-E.1, page 102.)

5.6.4 Words to avoid

"Instantaneous" in one of the above examples is one of a number of words, of which "user-friendly" is perhaps the most annoying, that are useful in ordinary language and may sound nice — who would argue *against* making a system "user-friendly"? — but are too vague to merit a place in requirements texts. Here is a list; it is not exhaustive but gives the idea. Each entry explains why the term or terms are bad and, if possible, suggests a replacement.

A checklist of terms that have no place in requirements texts	
Words to avoid	**Why (and how to replace)**
About, approximately, around	Subjective terms, not useful in requirements. Instead, give a tolerance range, e.g. "*5 cm ± 0.2 mm*".
Acceptable, appropriate, satisfactory, suitable	Subjective terms. Instead, explain what concrete properties a value, parameter, solution etc. must satisfy.
Certainly	If it is certain, no need to state that it is. If not, make it so and stop babbling.
Clearly, obviously	Noise terms, not useful in requirements. Making things clear is better than boasting about clarity.

Definitely	Noise term, not useful in requirements. Remove.
And so on, etc.	Requirements should leave no room for ad-libbing by the reader. See "Etcetera lists", 5.2.6, page 74.
Fast, immediate, instantaneous, real-time	Subjective terms. Instead, give hard bounds for response time.
Many, several, some	Give precise values or ranges if available. Even the simple variants "zero or more", "one or more" etc. are better.
Much, very	Subjective terms. Instead, give precise criteria. Or, remove altogether.
Should	Requirements are not the place for timidity. The system either must do something, or is not required to do it. "Shall" is the conventional terminology.
State-of-the-art	Vanity term, even though it may sound attractive in Goals documents. Instead, define applicable standards and technologies.
User-friendly	Vanity term. Instead, develop and document user-interface guidelines.
24/7	Sounds really impressive, but pure wishful thinking. Replace by precise requirements on incident handling and failsafe modes.

In many of these examples, the underlying criterion, among the quality factors of chapter 4, is *verifiability* and the associated concept of falsifiability (4.12). A requirement of "real-time" response is not verifiable. A precise specification of acceptable response times is.

5.7 STYLE RULES FOR NATURAL-LANGUAGE REQUIREMENTS

Writing requirements is, before anything else, writing, and should follow the general rules of good writing. But it is a specific kind of writing, entirely directed at the success of the project and resulting system, and driven by rules reviewed below.

5.7.1 General guidelines

Requirements can only be effective if their recipients — particularly programmers and testers — understand and trust them.

Gaining trust requires achieving a good level of quality. If developers come across sloppy requirements, they may be tempted not to pay them enough attention. They will second-guess or bypass the requirements, making instead their own requirements decisions, right or wrong.

The guidelines in the table on the next page are the principal rules of writing for requirements specifications. The basic justifications are in the table; further comments appear after it.

Basic rules of requirements writing	
Rule	**Comments**
Use correct spelling and grammar	Using correct language hardly costs more, and reinforces trust in the quality of the requirements. See 5.7.2 below.
Get the structure right	Avoid piling up a loose collection of individual requirements; instead, think about the overall structure of documents, to help the reader grasp each of the PEGS in its entirety. Chapter 3 gives the standard structure.
Use simple language	Other than technical terms from the domain, avoid any phrasing that will delay understanding for some readers.
Define all terms	You cannot expect all readers to know the domain vocabulary. In particular, beware of acronyms and include all relevant terms in the glossary (6.4).
Identify every part	Every element of the requirements, down to every paragraph, should have a unique identifier enabling convenient retrieval. See 5.7.4.
Be consistent	The same conventions should apply throughout the requirements specifications. For example, a given concept should always be denoted by the same term. See 5.7.5.
Be concise	Resist temptation to achieve clarity through redundant explanations.
Be precise	Imprecision causes misunderstandings and bugs.
Be prescriptive	The Goals, Project and System books enjoin developers and other project members to produce certain results. They should give clear prescriptions.
Separate explanation from prescription	Sometimes prescriptions require explanation. The two styles should be clearly distinguished.
Define responsibilities (use active style)	A passive style ("X will be done") leaves room for ambiguity. An active style ("Y will do X") avoids it.

5.7.2 Use correct spelling and grammar

There can be no justification for spelling and grammatical mistakes in requirements documents. Any text-processing tool will spot misspellings and suggest replacements. Any standard style guide (see the bibliography section) will remind you of the basic rules of good writing.

Even with these tools, mistakes will creep in; the way to produce a correct text is to make sure every element is reviewed by one or more people other than the writer. Requirements, as noted in one of our very first principles, are software (Requirements Nature Principle, 2.1.3).

Modern software development imposes strict procedures for *program* elements: no code is accepted until it has been reviewed by people other than its developer. Requirements elements deserve a similar process.

Pragmatic note from the projects in the author's experience: most of the failed ones did not apply this rule (they had sloppy requirements). Most successful projects did: they devoted comparable attention to the more mundane parts of requirements writing as to the substance of the requirements and to the rest of the development effort. Professionalism starts with the basics.

5.7.3 Use simple language

Requirements must be precise and to the point. Short sentences are usually sufficient. Other than technical terminology defined in the Glossary (6.4), it is generally appropriate to use ordinary words rather than their more fancy equivalents. In addition to the obvious merits of simplicity, note that many development teams are international; not all readers of the requirements will be native speakers of English (or another language with international reach).

Here are a few examples (among many) of preferring simplicity over pomposity:

- A basic mechanism provided by a system is a "function"; no need to glorify it into a "functionality". The latter word means a set of functions. (The system offers a certain overall functionality, consisting of a number of specific functions.) There is in fact seldom any reason to use "functionality" in the plural.
- In listing what a system can do, you can talk about its "capabilities", but the simpler word "features" usually does the job.
- No need to use such verbs as "facilitate" when "allow" and "support" are both simpler and more precise.

And so on. You will not be able to avoid scary technical terms from the problem domain (Environment) and information technology practices (Project), collected in the Glossary (6.4), but for explanatory and prescriptive elements of the requirements the simpler the words the better.

5.7.4 Identify every part

Identification Principle

Every element appearing in requirements must have a unique number or key allowing unambiguous identification.

Every element of the requirements must have a unique identifier. In particular:

- All structural text elements, starting with books (if the project follows the 4-book plan of chapter 3) and continuing with chapters, sections, subsections and individual paragraphs, must be indexed as part of a uniform numbering scheme, such as G, G.1, G.1.1 etc. for the Goals book. Some projects apply this rule down to the level of individual sentences.
- All other elements such as figures must also have individual identifiers.
- The Identification Principle applies to all other requirements-relevant items present in the requirements repository ("Requirements Repository Principle", page 26).

5.7.5 Be consistent

The quality of a requirements text does not just follow from the individual quality of its successive elements. Just as importantly, they must follow a consistent style. Discrepancies will confuse the requirements' consumers.

Already necessary for short requirements, this property becomes essential for the complex requirements that many industrial systems demand. They will usually be the result of the work of a team rather than a single person, making it likely that in the absence of a strict discipline inconsistencies will creep in.

As a simple example, English-language system requirements often use "shall-style" to specify behaviors: "*If the temperature sensor detects a temperature above the maximum, the system **shall** raise an alarm within no more than one second.*" Shall-style is attractive to many requirements writers because of its firmly prescriptive nature, reminiscent of military or legal orders. In truth, it is not indispensable; in many cases, "*must*" or other variants would also work. But whichever one a project chooses, it should stick to it. Changing the convention from one part of the requirements to another confuses readers and can lead to misunderstandings.

The same rule applies to all aspects of documents, including more mundane properties of style; as two examples among many:

- If some bulleted lists are numbered, all bulleted list should be numbered.
- In English, choose American or British spelling, but do not use *color* (US) in one place and *catalogue* (British) in another.

One place where good requirements writing differs from good non-technical writing is the matter of repetition. Not repetition of the *properties* themselves specified in the requirements — the "No-Repetition Principle", page 76, warned us against this practice — but repetition of *words* in the text, as a style issue.

At school, we were told to vary terminology to avoid boring the reader. If you wrote that the mood was dark and want to insist, you might, according to such advice, call it "somber" the second time around. Well, maybe in a novel, but not in a requirements document. The terminology must be both precise and consistent. If (in the environment of your project) a vat is the same thing as a tank, call it a vat or call it a tank but stick to one name throughout. The developers reading the requirements as a guide to design and implementation need that consistency.

Synonyms do exist even in technical domains. It may be the case, for example, that some technical documents beyond your control, describing the environment of your project, use "vat", and others use "tank". What the requirements documents should do here is clear: help avoid the ambiguity by using one term only, but devote an entry of the Glossary to the other, as in: "*Tank: synonym for vat*", with a hyperlinked reference. (Nothing prevents you, in the "vat" entry, from adding "*also called tank*".)

5.7.6 Be prescriptive

The meek may be destined — albeit presumably long-term — to inherit the earth, but they do not write good requirements. Requirements should be firm, not timid. (Correction: they *shall* be firm.)

For the Project, state without hesitation what the tasks, assignments and milestones are; for the Environment, define the relevant properties with precision; for the Goals, make it clear what the organization expects; for the System, specify the expected behaviors.

5.8 THE TBD RULE

TBDs ("To Be Determined"), elements left out for later completion, are a plague of requirements documents. Strive to avoid them.

TBDs can be just an excuse for laziness and procrastination, leaving out some properties because they are hard to specify. They can be legitimate when they reflect that some information is not known at the time of writing and expected to become available later. Or that the requirements writers simply decided for now to focus on some elements and leave others to later.

That some property is "to be determined" does not mean, however, that we should say nothing about it. All too often one finds cursory and careless TBDs, as in example:

Triggering conditions for overheat alarm: TBD.

Such a form is unacceptable in a professional setting. TBDs must adhere to the following rule:

TBD rule

Any incomplete ("To Be Determined") mention in requirements must include:

1 Name of author declaring the property "tbd".
2 Date the property was found to be "tbd".
3 Date or project phase by which the indetermination should be resolved.
4 Importance of resolving it, one of: show-stopper, serious, desirable.
5 What will be needed to resolve it, one or more of: stakeholders to ask; documentation to consider; management decision (by whom?).

In addition, the requirements must include a **TBD list** with links to all TBDs.

For example:

Triggering conditions for overheat alarm: TBD
1. Introduced by: Bertrand Meyer
2. On: 2021-09-10
3. Importance: serious
4. Resolve before: start of any coding of the alarm management module.
5. To resolve:
Stakeholders: heating control engineers
Documents: heating system manual (version 4, expected 2021-10-10).

It takes only minimal effort to produce such a sketch of what the missing part will look like, but doing so systematically is essential to the quality of requirements. It is also a matter of courtesy: giving the consumer of the requirements an idea of why something was left out and what you intend to do about it.

The TBD list — similar to such "metarequirements" elements (1.9.4) as a table of contents or table of figures at the beginning of a requirements document — can be generated automatically by the text-processing system. As a side benefit, it provides a useful indication of the state of completion of the requirements effort and its progress.

5.9 DOCUMENTING GOALS

The rules discussed so far in this chapter apply to all four PEGS. The Goals book has distinctive features (3.4, page 38). It addresses a broader audience, including high-level management and others who need to be informed of the project's overall scope and purpose but:

- May not directly participate in the project, other than to approve its launching and determine acceptance of its intermediate and final results.
- May not possess specific technical IT or subject-matter expertise.

The Goals book should be adapted to this audience and its needs. In particular:

- It should be short (thirty to sixty pages is typical).
- It should convey general ideas, not technical details.
- It should take the perspective of the enterprise, not the project, including benefits (the topic of its G.3 chapter) and limitations (G.3).
- It should emphasize clarity and readability, making graphical representations (5.5.2) particularly appropriate in many cases.

Emphasizing clarity does not mean sacrificing integrity. The picture given in the Goals book may be simplified, but should still be accurate. The book should in particular resist the temptation to gild the picture or over-promise. The backlash would inevitably come.

5.10 THE SEVEN SINS: A CLASSIC EXAMPLE

To conclude this discussion of how to write good requirements, we go back to a small example that has figured in several classic papers. You will find the references in the bibliography section of the present chapter (page 103).

The example's origin is an article on proofs of program correctness by a well-known computer scientist, Turing-Award winner Peter Naur. Two researchers on software testing (another approach to verification), John Goodenough and Susan Gerhart — G&G below — criticized the paper and stated in passing that part of the problem was a poor requirements specification; in their paper, they offered a replacement. A third paper, by the present author, critically analyzed their own specification. What follows is a thoroughly updated version of that analysis.

We will take up the problem again when discussing formal methods in chapter 9, providing in 9.5 a mathematical specification and a new English-language specification derived from it.

The problem under examination — splitting a text across lines — is very small, far from the requirements complexity of modern software systems. This simplicity is part of what makes the example fascinating: it serves as a microcosm of much of what can go wrong in requirements writing. It is almost scary: if there is so much potential for messing up in a small text-pro-

cessing case, what about a real industrial project? The reassurance is that it is indeed possible to write good requirements, large or small, by following the principles of this Handbook. For a large example, see the Companion to this Handbook, which applies the principles to a significant industrial system.

A warning about the text-formatting example

This example is a simple problem. You may have the impression that the following discussion (and the further formal treatment in 9.5) belabors it far beyond its significance. If you feel that way, jump to the next chapter at the first sign of boredom. But do come back later. However small and seemingly obvious, the example is full of surprises and of lessons which apply to requirements of systems, all the way up to the very large, and are much more vividly highlighted on the very small.

5.10.1 A simple specification

The problem description in Naur's original program-proofs article was as follows:

The Naur specification

Given a text consisting of words separated by BLANKS or by NL (new line) characters, convert it to a line-by-line form in accordance with the following rules:

1 Line breaks must be made only where the given text has BLANK or NL;

2 Each line is filled as far as possible as long as

3 No line will contain more than M characters.

(In Naur's paper, M was called "MAXPOS". Other than this change meant for brevity, the Naur and G&G specifications in the present discussion are verbatim citations from the originals.)

The intent is clear: the program reformats a text by re-splitting it over lines filled as much as possible. With the example input on the left below, and M = 9, the output might be as on the right. (BLANK characters represented as "␣" for readability; NL shown by starting a new line.)

Input	Output (M = 9)
1 5 10 15	1 2 3 4 5 6 7 8 9
T O ␣ ␣ B E ␣ ␣ ␣ ␣ O R ␣ ␣ N O T	T O ␣ B E ␣ O R
␣ ␣ ␣ ␣ T O ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣	N O T ␣ T O ␣ B E
␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ B E ␣	

With the understanding that it was not written as a model requirements text, but as a short problem description in an article devoted to another topic (program proofs), it is a useful exercise to assess this requirements specification, asking yourself for example:

- Does it clearly describe the problem?
- Is there always a solution?
- Can there be more than one solution?
- Does it say too little (ambiguity)? Too much (noise, overspecification)?

5.10.2 A detailed description

Goodenough and Gerhart criticized Naur's program-proving approach on a number of grounds and, interestingly for this discussion, mentioned that part of the problem was a poor specification. They provided their own replacement, which we will analyze.

The Goodenough-Gerhart specification

The program's input is a stream of characters whose end is signaled with a special end-of-text character, ET. There is exactly one ET character in each input stream. Characters are classified as:

- Break characters - BL (blank) and NL (new line);
- Nonbreak characters - all others except ET;
- the end-of-text indicator — ET.

A word is a nonempty sequence of nonbreak characters. A break is a sequence of one or more break characters. Thus, the input can be viewed as a sequence of words separated by breaks, with possibly leading and trailing breaks, and ending with ET.

The program's output should be the same sequence of words as in the input, with the exception that an oversize word (i.e. a word containing more than M characters, where M is a positive integer) should cause an error exit from the program (i.e. a variable, Alarm, should have the value TRUE). Up to the point of an error, the program's output should have the following properties:

1 A new line should start only between words and at the beginning of the output text, if any.
2 A break in the input is reduced to a single break character in the output.
3 As many words as possible should be placed on each line (i.e., between successive NL characters).
4 No line may contain more than M characters (words and BLs).

The text appears again on the next page, with some of its deficiencies — the "seven sins" — highlighted. Before turning (or scrolling) the page, though, it is useful to go through the text yourself and assess its quality. Did you find anything suspicious? You can compare your results with the following analysis.

The Goodenough-Gerhart (G&G) specification (annotated)

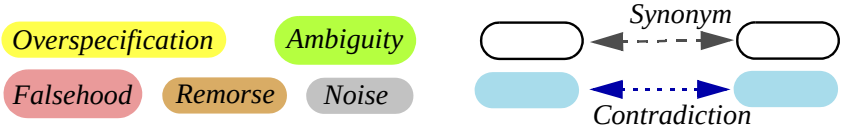
The program's input is a stream of characters whose end is signaled with a special end-of-text character, ET. There is exactly one ET character in each input stream. *1 2*
Characters are classified as: *3*

- Break characters - BL (blank) and NL (new line); *4*
- Nonbreak characters - all others except ET; *5*
- the end-of-text indicator — ET. *6*

A word is a nonempty sequence of nonbreak characters. A break is a sequence of one or more break characters. Thus, the input can be viewed as a sequence of words separated by breaks, with possibly leading and trailing breaks, and ending with ET. *7 8 9*

The program's output should be the same sequence of words as in the input, with the exception that an oversize word (i.e. a word containing more than M characters, where M is a positive integer) should cause an error exit from the program (i.e. a variable, Alarm, should have the value TRUE). Up to the point of an error, the program's output should have the following properties: *10 11 12 13 14*

1 A new line should start only between words and at the beginning of the output text, if any. *15 16*
2 A break in the input is reduced to a single break character in the output. *17*
3 As many words as possible should be placed on each line (i.e., between successive NL characters). *18 19*
4 No line may contain more than M characters (words and BLs). *20*



That text is a festival of the "seven sins" of the requirements writer. The critique is justified since the specification was not just quickly jotted down as an adjunct to the discussion of a non-specification problem, but presented as a response to the bad quality of the original Naur problem statement. We take the problems in the order of the text, numbers in parentheses referring to its line numbers. For clarity, we represent a BLANK (space) character as ␣.

(*1*) and (*10*) introduce a **contradiction**. The input is not a "sequence of words" (*10*) but a stream of characters. These are different kinds of entities: ['T', 'O', '␣', 'B', 'E'] is a sequence

of five elements (characters) whereas ["TO", "BE"] is a sequence of two elements (strings). The statement that "*The program's output should be the same sequence of words as in the input*" is interesting (less politely, it is a **falsehood**) since in fact **neither the input nor the output** is a sequence of words; they are both sequences of characters. True, (*8-9*) states that "*the input **can be viewed as** a sequence of words separated by breaks*" but this is only an explanation (noise, as will be seen below); saying that A "can be viewed as" B does not mean that A is B.

(*1*), (*2*), (*7*) and (*10*) introduce **synonyms**: "stream" and "sequence" mean exactly the same. "Sequence" is the better term, since it has a precise mathematical definition (see "Sequences", 9.2.6, page 166); "stream" is more vague. Requirements must use a single term for every notion (No-Synonym Principle, page 74: keep names and concepts in one-to-one correspondence).

The introduction of the ET character (*1-2*, *6*) is gratuitous **overspecification**. True, the C programming language terminates strings with a special character, a "null", written \0; and in some file systems a sequential file has an end marker. Such conventions, however, are pure implementation matters. The ET character was not part of the original Naur specification; it is not necessary for understanding the problem (or for implementing a solution, since any programming language or library mechanism for reading files will make it possible to read characters until the end without having to know how end-of-file is marked internally); it seriously complicates the specification; and it leads to other problems.

One of these other problems comes out in the next sentence (*2*) after the introduction of ET. "*There is exactly one ET character in each input stream*" is pure **noise** since it has already been specified that ET is a "*special*" and "*end-of-text*" character. Clearly, since a sequential file has exactly one end, if ET is the end-of-character, there can only be one ET. The extra precautionary explanation wastes readers' time (or makes them raise unnecessary questions). The word "*special*" is itself noise: what makes a character "special"? Saying that ET is "end-of-text" is enough to make it special. See how overspecification, far from helping, creates an urge for *explanation* (5.4), which in a seemingly endless pursuit only causes new intellectual contortions.

Another **synonym** case is "*nonempty*" sequence of characters (*7*) versus "*one or more*" characters (*8*). The two mean the same thing but in requirements the change of phrasing can cause confusion. (No-Synonym Principle again.)

"*Oversize word*" (*11*) is **noise** since the concept is defined precisely next (same line) and the term "oversize" is not used in the rest of the specification. There was no need to introduce it.

"M *is a positive integer*" is **remorse**: M is a parameter of the problem and should have been introduced at the beginning, not after the concept has been used. The phrasing is also a case of **ambiguity**: we have to understand that M applies to the processing as a whole, rather than being defined in each case. In other words, the statement "*a word containing more than* M *characters, where* M *is a positive integer, should cause an error exit*" could be interpreted to mean that every word of length 2 or more causes an error exit, since such a word has the property that it is "*more than* M *characters, where* M *is a positive integer*" (take M = 1). Proposing such an interpretation is playing silly, of course, but only because we know about the problem domain (Environment): we all have at least an intuitive understanding of the kind of text pro-

cessing involved. Consider now the case of a developer interpreting requirements specification for a domain in which he or she is not an expert: true confusions and misunderstandings can result. These problems are entirely avoidable by taking a systematic approach to describing the problem: define each element exactly once; rather than explaining, make the definition precise.

The whole idea of introducing (*12-13*) "*a variable, Alarm*" that "*should have the value TRUE*" is pure **overspecification**. It is not the business of a problem description to command the programmers to use particular variables. Specify error cases and (at the appropriate level of detail) the processing for each of them.

The requirement that the program's output should have the required properties "up to the point of an error" (*13*) is a case of **ambiguity**: it does not determine whether an oversize word should be partially output or not. Assume M = 6 and the input

THAT␣IS␣THE␣QUESTION

so that the first two lines of the output can be

THAT
IS␣THE

What, however, is "the point of the error"? We can take it to be the beginning of the oversize word, in which case the output stops here; or the first offending letter, in which case we must output a third line with M characters:

QUESTI

With the text as given, either of these interpretations is as plausible as the other — in this case not even depending on familiarity with the problem domain. (Plain ambiguity: neither common sense nor text-processing expertise helps.) The situation is typical of many requirements documents which devote great care to unimportant matters, resulting in noise and overspecification, while remaining silent on important properties, which developers need to know.

"*A new line should start only between words and at the beginning of the output text*" (*15-16*) introduces **ambiguity**. The phrasing is infelicitous in any case: it is not that a new line "should" appear between words and at the beginning (otherwise there would be new-line characters between all words!); the text should have said that it "*may only*" appear there. But — worse — the sentence also suggests that it is OK to add a new-line character at the beginning of the text. The problem here is that the G&G specification does not define what it means by "*new line*", causing confusion since this term conflicts with a defined concept, the new line or NL *character* (*4*). The authors are relying on the reader's intuition of what a *line* is. When they write that a "*new line*" should "*start*" they are not just talking about the characters following an NL character, but also about the first line, usually not preceded by NL. This intuitive understanding, however, is not explicitly specified, and creates a contradiction with concepts that are.

Never make such an assumption in a requirements document; define all concepts (Glossary Principle, 6.4, page 107). It is particularly ironic here that, as we will shortly see, the authors do attempt a few lines down (*18-19*) to define (if not "*new line*") the notion of "*line*", and manage to get it wrong!

"*The output text, if any*" (*15-16*) is **remorse** (5.2.3). So far the reader has been led to assume that there would be an output. That seems like a matter of course: a text-formatting program takes some input and produces some output. We were told, unambiguously, that (*10*) "*The program's output should be*" a sequence of words. If it is (or "*should be*") something, it must exist. But now, 15 lines into a 20-line specification, comes the news that it might not. Why? Under what conditions? No clue. To the programmer trying to implement a solution, the mystery is not reassuring.

The requirement (*18*) that "*as many words as possible*" should be put "*on each line*" is a new case of **ambiguity**, but of a different nature from the preceding ones. With the input text

TO␣BE␣OR␣NOT␣TO␣BE

and M = 8, all of the following candidate outputs satisfy the other conditions and have the same number of lines (3, the minimum possible):

```
1 2 3 4 5 6 7 8
T O ␣ B E ␣ O R
N O T ␣ T O
B E
```

(A)

```
1 2 3 4 5 6 7 8
T O ␣ B E ␣ O R
N O T
T O ␣ B E
```

(B)

```
1 2 3 4 5 6 7 8
T O ␣ B E
O R ␣ N O T
T O ␣ B E
```

(C)

This example shows that it is impossible to ensure that the output fits "*as many words as possible on each line*" for *all* lines: (A) and (B) fill up the first and second lines as much as possible, but not the last one; (B), the first and last but not the second one; (C), the second one but not the others. Seen that way, the specification is a case of **falsehood** rather than ambiguity. If we accept that it was really intended to mean "the number of lines should be the minimum possible", then it is not a falsehood, but we have to deal with the ambiguity: in such examples, which of the three variants does it define as acceptable?

The indication that the text should satisfy some properties "*up to the point of an error*" (*13*) seems to suggest that it was written under the assumption that processing would be purely *sequential*, filling each line as much as possible before proceeding to the next line. Under this interpretation, there is only one suitable output in the example: (A). Is also possible, however, that the authors intended to allow any output meeting the criteria; then (A), (B) and (C) are all acceptable results. A requirements specification allowing several solutions is called *non-deterministic*. If that was the intention, the text should have explicitly mentioned the non-determinism, a much more interesting property than the included noise and overspecification elements.

"*Each line (i.e., between successive NL characters)*" (*18-19*) is again **remorse**, defining in the penultimate line a concept already used before. It is also something worse: a **falsehood**. It is simply wrong that a line is always delimited by two new-line characters: consider the first and last lines, both of which are flanked by only one NL, respectively after and before. This wrong definition reflects absent-mindedness in the writing of the specification, but imagine again a problem domain where the programmer's intuitive understanding is of no help. Programmers may follow the definition and implement the wrong behavior.

The final statement in the text (*20*), "*no line may contain more than* M *characters (words and BLs*", is yet another **falsehood**: a word is not a character! A word is a sequence of characters (*7*). There are several ways to express the intended property, but the best one is simply to remove the useless and ultimately damaging attempt at explanation: "*(words and BLs)*". Just stating "*no line may contain more than* M *characters*" is precise and definitive. We see here one more attempt at explanation that brings no good and causes harm.

It is indeed fascinating to see how the last two falsehoods, and many of the other deficiencies identified in the previous discussion, follow from worthy intentions gone sour. The attempt to explain "*word*" peters out as badly as the attempt to explain "*line*". This phenomenon is characteristic of requirements documents gone awry. Remember the Explanation Principle: specification beats explanation. *Say it once, say it well*.

5.10.3 More ambiguity!

At this point you might think that we have squeezed the text-formatting example more thoroughly than any single lemon in the entire world history of lemon-squeezing and split it more thinly than any single hair in the entire world history of hair-splitting. The analysis of the G&G specification cannot possibly have missed any flaw, however minute. And yet it did! A pretty important ambiguity at that. In fact, two ambiguities.

In all the preceding example solutions, lines start with a letter. Now assume (still for M = 8) that the input is ␣TO␣BE␣OR␣NOT␣TO␣BE, as earlier but with an added initial space (or several initial breaks). Then none of the previous solutions is correct any more, since they do not start with a space and hence, for that character, violate the condition that "*A break in the input is reduced to a single break character in the output*" (*17*). We may contract breaks (indeed we must), we may not remove them. A correct solution is:

```
␣TO␣BE
OR␣NOT
TO␣BE
```

The same phenomenon occurs if the text (ignoring the spurious "ET" business) *ends* with one or more break characters: they have to yield a trailing space.

There is neither contradiction nor falsehood here: this interpretation of the problem makes sense. It looks suspicious, however, since text formatting of the given style typically gets rid of ending and trailing breaks altogether. Formatted text does not generally start or end with spaces. The ambiguity comes very close to a contradiction in light of "*As many words as possible should be placed on each line (i.e., between successive NL characters)*" (*18-19*), already flagged for other problems. While not explicitly stating it, this phrasing strongly suggests that for inside lines the first and last word are directly flanked by NL characters, without any intervening space. If so, the property would presumably apply also to the first and last lines, although (as we have seen) the statement mistakenly did not take them into account.

By the way, what output shall we produce for an input made of break characters only: an empty text, or one consisting of a single space? Not the most poignant of all dilemmas, but in the requirements of more significant systems such ambiguities could have serious consequences.

We have reached the stage of making conjectures about the requirements authors' intent; that should never be the case with well-written requirements. It is possible, although unlikely, that the intent was indeed to preserve a leading or trailing break, but at the very least (if only because this convention is counter-intuitive) the specification should then have stated it explicitly, removing any doubt in the minds of readers and particularly of implementers.

5.10.4 Lessons from the example

In light of the preceding analysis it is interesting to compare the two specifications. G&G criticized the Naur specification (page 94) and explicitly presented the revision (page 95) as an improvement. Is it?

The original did have a serious flaw: the absence of any mention that the problem only has a solution if no word is longer than M. Other than the need to specify this condition (which is easy to add), it was simple and immediately understandable. In its effort to leave no stone unturned, the revised version entangles itself in contradiction after complication and ambiguity after incongruity. Every attempt at clarification results in added noise or worse, while leaving open some real questions, including:

- In the case of an oversize word, **how much** of the input text must we process?
- Do we discard a **leading or trailing break**, or do we keep it, and if so in what form (such as a single space)?
- Must the program fill lines as much as possible in the order of the input text (**sequential** processing), or is it possible to fill later lines more than earlier ones (for example to produce a more pleasing visual effect) as long as the total number of lines is minimum?
- More generally, can the problem admit more than one correct output (**non-determinism**)?
- Is there always **at least one** correct output?

The extra length of G&G as compared to Naur (a ratio of 4) seems hard to justify with so many questions left unanswered.

The lesson for requirements writers is clear: do not attempt explanation (except in the few places where it might be essential according to the list of 5.4); ruthlessly cut noise; spot and remove ambiguities; constantly aim for precision; and remember ("Requirements Questions Principle", page 22) that one of the key aims of a requirements specification is to uncover all important questions about the problem and answer them.

5.10.5 OK, but can we do better?

With the extensive criticism leveled so far at the G&G text, you are entitled to a switch from the negative to the positive. What would be a satisfactory version of this specification, immune to such criticism?

Such a replacement is indeed coming, but it will take advantage of a detour through the tools of mathematical reasoning, the key to precision. Chapter 9, devoted to formal methods, takes up the problem again in a section entirely devoted to it ("An example: text formatting, revisited", 9.5, page 171). It first develops a mathematical model; then, coming back to English with the benefit of that model ("Back from the picnic", 9.5.7, page 178), it proposes a new natural-language version, quite different from anything we saw in the present chapter. You are encouraged to analyze it — just as critically as this discussion did for the G&G version — for its adherence to quality factors for requirements, and its avoidance, or not, of the Seven Sins of the Specifier.

5-E EXERCISES

5-E.1 Addressing criticism

Sections 5.6.1, 5.6.2 and 5.6.3 provide three examples of proposed replacements for deficient phrasings of the requirements, then have some criticism of the replacements themselves. Using if necessary your intuitive understanding of the problem domain in each case, provide your own replacements addressing the criticism. Include a discussion of the issues involved and a justification for your rewrites.

5-E.2 Text formatting

Propose a better version of the text-formatting requirements of section 5.10. Submit it to critical analysis and compare it to the two given there. (Note: preferably do this exercise before reading chapter 9, which discusses the example further and proposes both a formal replacement and a non-formal one. You may want to go back to your answer after reading that chapter, and consider whether it needs updating in light of the concepts introduced there.)

5-E.3 Do not just format, justify

Adapt the specification that you obtained in exercise 5-E.2 to a variant of the problem in which output text is left- and right-justified through the possible addition of blank characters.

Evaluate how much you had to change, as a clue to the *modifiability* (4.11, page 65) of your original specification.

BIBLIOGRAPHICAL NOTES AND FURTHER READING

Anyone writing requirements and other technical texts should read and apply the precepts of the soon-centenary *Elements of Style* [Strunk-White].

The examples and criticism of bad specification elements are from [Wiegers-Beatty 2013], which presents more rules of requirements style.

A discussion of the Single Point of Maintenance rule (page 76) appears at [IfSQ 2016].

The network architecture diagram of page 80 is from [Wikipedia: SOA] and the diagram illustrating table notations on page 83 from [Wikipedia: OSI], both Wikipedia pages.

The "seven sins of the specifier" (5.2) are an updated version of those in [Meyer 1985]. That article also discussed the Naur [Naur 1969] and Goodenough-Gerhart [Goodenough-Gerhart 1977] specifications discussed at length in the present chapter.

[Bruel et al. 2021] is a survey of notations for requirements (5.5), particularly formal notations (5.5.3). It also includes a discussion of tools that use a restricted version of natural language (as mentioned at the end of 5.5.1).

David Parnas has promoted the systematic use of tabular specifications (5.5.4); see [Parnas 2001] (and a summary and assessment of the approach in [Bruel et al. 2021]).

The concept of "multirequirements", the combined use of multiple notations for requirements, is discussed in [Meyer 2013].