Requirements Eng (2004) 9: 67–82
DOI 10.1007/s00766-003-0181-1

ORIGINAL ARTICLE

D. M. Berry · K. Daudjee · J. Dong · I. Fainchtein
M. A. Nelson · T. Nelson · L. Ou

User's manual as a requirements specification: case studies

Received: 21 May 2001 / Accepted: 15 July 2003 / Published online: 22 November 2003
© Springer-Verlag London Limited 2003

Abstract This paper argues that a user's manual makes an excellent software requirements specification. It describes several experiences, including one in industry, of writing user's manuals as requirements specifications. Finally, it discusses several lessons learned from the experiences.

Keywords Requirements elicitation · Requirements specification · Requirements validation · Scenarios · Test cases · User's manual

1 Introduction: the problem of writing good requirements specifications

There are a number of reasons that writing a requirements specification (RS) for a computer-based system (CBS[1]) before implementing it is a good idea [12, 45, 4, 46, 33, 48, 36].

1. The process of writing the RS of the CBS is a good way to learn the CBS's requirements.
2. The process of writing the RS of the CBS helps to reconcile differences among the CBS's stakeholders.
3. The RS allows the customer of the CBS to validate that the projected CBS will be what he[2] wants before resources are spent implementing a possibly incorrect CBS.
4. The RS makes it clear what must be implemented to obtain the required CBS.
5. The RS allows deriving both covering test cases and expected results that allow verification that the implementation of the CBS does what it is supposed to do.

Despite the clear benefits of writing an RS for a CBS before implementing it, many projects are unable to produce the RS, for a variety of reasons, some technical and some social.

1. It is difficult to write a good RS, one that specifies exactly what the CBS is supposed to do without limiting unnecessarily how to implement it.
2. Participants in most projects these days believe that they do not have the time to do so, that it is necessary to proceed immediately, if not before, to coding, in order to meet the code's delivery deadline or to be the first in the market with the code's functionality (begging the question of how do they know what to implement anyway if requirements are not specified).
3. Participants in most projects these days perceive that time spent on writing an RS is wasted since the requirements will change anyway and the RS will probably never be read, even by the implementers.

This paper offers the writing of a user's manual for a CBS before implementing it as a method of achieving the writing of an RS of the CBS before implementing it. The method both

– produces a document that delivers the five benefits of writing an RS before implementation and

D. M. Berry (✉) · K. Daudjee · J. Dong · I. Fainchtein
M. A. Nelson · T. Nelson · L. Ou
School of Computer Science,
University of Waterloo,
Waterloo, Ontario,
N2L 3G1, Canada
E-mail: dberry@uwaterloo.ca

[1] The focus of building a CBS is on writing the software. Hence, often we forget that we are dealing with a whole system and talk about developing software. Moreover, as we give requirements for a CBS, we also need to give requirements for the software component in what is known as a software requirement specification (SRS). In this paper, fully cognizant of the necessity to deal with the whole system, we often use "system" and its "software" interchangeably, particularly since the part of the system that is most malleable is the software.

[2] To avoid the clumsy "he or she" construction, which in turn avoids using plural "they" for singular "everybody", any unspecified person on the customer's or user's side of the game is "he" and any unspecified person on the analyst's or implementer's side of the game is "she".

– helps mitigate the three problems that discourage the production of an RS before implementation.However, the benefits and problem mitigation accrue only for those CBSs for which a user's manual is possible, i.e., the CBS has users and a user's manual that describes essentially all of the CBS's functionality. Therefore, unless the applicability issue is at hand, we assume that the CBS being developed and specified is appropriate for offering a user's manual as its specification.

Accordingly, Sect. 2 shows how a user's manual can serve the five listed purposes of an RS and cites other work that agrees with this claim. Section 3 shows how production of a user's manual may help avoid the three problems that inhibit the production of an RS. Section 4 describes the structure and contents of a good user's manual, and notes that they are the structure and contents of a good RS. Section 5 details the kinds of CBSs for which user's manuals provide good coverage of their requirements and describes how to deal with requirements that are not appropriate for presentation in a user's manual, including nonfunctional requirements. These sections only argue their points. It is necessary to validate these claims in practice. Accordingly, Sect. 6 presents four case studies of projects, three academic and one industrial, in which user's manuals were used as RSs. All four involve the production of substantial software applications. These case studies support the claims of Sects. 2 and 3. Section 7 lists lessons learned in these case studies, beyond those that support the claims. Section 8 concludes the paper with a discussion of threats to the conclusions and of future work to test the validity of the lessons learned.

As is shown in Sect. 2.1, the idea of considering a user's manual an RS is quite old. The contribution of this paper is not the idea but the validation of the idea. The contribution of this paper is its descriptions of the details of several developments of nontrivial CBSs, developments in which the user's manual played the role of the RS. These case studies allow evaluation of the effectiveness and the payoff of the approach. Demonstrated effectiveness and payoff can help provide the motivation that overcomes the inhibitions against writing an RS early in a CBS's development.

2 User's manuals as requirements specifications

This section argues that for an appropriate CBS, a good user's manual has most of the information that is needed in an RS that focuses on the what, as opposed to the how, of the described CBS. Therefore, a good user's manual can satisfy the five listed purposes of an RS.

2.1 Information in a user's manual

A bit of thought reveals that the information that is in a properly written user's manual for a CBS is precisely what should be in an RS of the CBS. We are told that the RS should describe what the CBS does and not how the CBS does it [18, 24, 5]. We are told that the RS should describe the CBS's function and not the CBS's implementation. We are told that the RS should describe the CBS from the user's point of view and not the implementer's. A good user's manual for a CBS describes what the CBS does and not how the CBS does it. A good user's manual for a CBS describes the CBS's function and not the CBS's implementation. A good user's manual describes the CBS from the user's point of view and not from the implementer's.

In fact, as early as 1975, in *The Mythical Man-Month* [9], Brooks implicitly equated the manual with the written RS for a computer system product by describing the manual's mission as that of an RS.

> The manual must not only describe everything the user does see, including all interfaces; it must also refrain from describing what the user does not see. That is the implementer's business, and there his design freedom must be unconstrained. The architect must always be prepared to show *an* implementation for any feature he describes, but he must not attempt to dictate *the* implementation.Also, de Marco suggests in several places using user's manuals as RSs, most notably in *The Deadline* [13].

Urban legend has it that the user's manuals for the Lisa and Macintosh computers were written completely before implementation of the software began. These manuals were then given to the system programmers as the specification of the appearance and functionality of the user interfaces and hence of the underlying systems [2].

2.2 The five purposes of a requirements specification achieved by writing a user's manual

This section argues that the process of writing a user's manual and the resulting user's manual serve the five purposes of writing an RS for a CBS before implementing the CBS.

Writing a good user's manual for a CBS requires a clear conception of what the CBS is supposed to do, clear enough that the manual's author can visualize scenarios [49, 10] of the use of the CBS and describe both

1. what the user should say to the CBS and
2. what the CBS should respond to the user

in each of these scenarios. Describing this information amounts to describing the use cases of the CBS [21, 10, 11], because scenarios, with steps in common collected into user-task-focused abstractions, are none other than use cases. Thus, the very process of determining what to say in a user's manual is the same as the process of eliciting requirements to write an RS.

If a stakeholder says that the user's manual for a CBS describes the desired functionality for the CBS, the stakeholder has effectively validated that the CBS

described is the CBS the stakeholder desires. Moreover, when different stakeholders have conflicting requirements, these conflicts manifest themselves as differing reactions to portions of the user's manual. Discovery of these differing reactions can be followed by negotiations to resolve the conflicts [7].

Any time an implementer has a question about the functionality of a CBS, she can consult the CBS's user's manual to determine the expected behavior of the CBS, just as a user consults the CBS's user's manual to determine both what he can say to the CBS and the CBS's expected response to what he has said.

An RS is often accompanied by or includes descriptions of scenarios and use cases. These should be the same scenarios and use cases that describe how users exercise the CBS to do their work. These scenarios and use cases in turn form a good basis for building test cases that cover the expected ways the CBS will be used [29]. Moreover, since the users are guided by the user's manual in their uses of the CBS, these test cases provide a good coverage of the expected uses of the CBS.

Finally, if there are different kinds of users of a CBS, e.g., ordinary application users and system maintainer users, then a different user's manual should be written for each kind of user, addressing the way he sees the CBS. These user's manuals specify differing, possibly overlapping, sets of functions for the CBS.

3 Motivating the writing of a requirements specification

Writing a good RS for a CBS is difficult, it delays getting on to coding, and it is considered a waste of time. This section considers several ways to motivate writing an RS for a CBS and overcoming these inhibitions [29]. Some, but not all, of these ways involve writing a user's manual as the RS.

Writing a good RS for a CBS is difficult because the advice to describe only what the CBS does, without constraining how the CBS does it, is easier said than done. Clients and users tend to describe solutions to possibly nonexistent problems rather than just problems that need to be solved [18]. Consequently, requirement analysts who are not domain experts tend to think about the CBS in terms of solutions and treat a specific solution as a general requirement. They produce RSs specifying many implementation details. These details are mixed in with general requirements that are offered as rationale motivating the solutions disguised as requirements.

On the other hand, writing a good user's manual for a CBS forces focusing on the user's view of the CBS. With the typical user in mind as the future audience of the user's manual, it becomes easier to focus on the user's view, the what of the CBS, and to avoid burdening the user with implementation details. Even if a requirements writer should lapse into writing about some implementation details, a test audience of future users will surely complain about these extraneous digressions into implementation details. Thus, while a user's manual is ostensibly different from an RS, they in fact convey the same information, and a good user's manual should be easier to write than a good RS.

The other inhibitions against writing an RS for a CBS are that:

1. In many organizations, the perception is that there is not enough time to write the RS; it is necessary to get on to the coding as quickly as possible.
2. Many view writing an RS as unnecessary, as a waste of time. After all, the requirements will change as the program is being written, and the RS will not be kept up to date; therefore, why bother?

One possible way to motivate writing an RS is to foster a realization that the RS gets written anyway, as one is writing the user's manual and the help system and as one is devising test cases, particularly if it is important to deliver a quality CBS product [27]. Writing the user's manual requires determining what the CBS does, i.e., its requirements. Also, in order to test a CBS, a full set of test inputs must be determined. This determination requires figuring out what the CBS is supposed to do, i.e., the full set of features. Then it is necessary to figure out, for each test input, the expected output. This figuring requires determining what the CBS is to do for each test input. The complete test plan amounts to an RS of the CBS, covering at least the tested features. The RS is as complete as the test cases.

Indeed, it is clear that in any CBS development other than for totally private use, if the normal minimum required to release and sell the CBS is done, there is a budget for testing and writing the manual or help system. Therefore, there are enough resources for producing an RS. These resources include the time for testing and writing the user's manual or help system. Therefore, it is simply not true that there is not enough time to write the RS. Moreover, since an error discovered at requirements specification time costs two orders of magnitude less to fix than the same error discovered at delivery time [6], it pays to write the RS, test cases, and user's manual or help system at the beginning of the project.

It is clear that writing an RS is essential. However, why is the user's manual more attractive than an RS? The user's manual needs eventually to be written anyway for future users' benefit, while the RS is likely not to be looked at beyond the beginning of coding. Also, a user's manual almost always exposes a full set of use cases simply because the manual's purpose is to show how to use the CBS, and those use cases can then be used to generate the essential test cases. On the other hand, an RS written according to at least one of the standards for SRSs [23] tends not to expose a full set of use cases, called *modes of operation*, because the SRS's focus is on the functions to be implemented by the CBS.

4 Contents of a good user's manual

Our own informal examination of a variety of user's manuals leads us to believe that a good user's manual has the following elements:

1. descriptions of underlying and fundamental concepts of the CBS,
2. a complete graduated set of examples, each showing
   - a problem situation the user faces,
   - some possible user responses to the problem in the form of commands to the CBS, and
   - the CBS's response to these commands, and
3. a systematic summary of all the commands.

The descriptions of the underlying and fundamental concepts of the CBS constitute a lexicon for the CBS and its requirements [34]. The complete graduated set of examples constitutes a defining set of use cases for the CBS [25].

Having only the third loses many readers who do not understand the concepts and turns off many readers who get bored reading page after page after page of command syntax and semantics. Leaving out the first makes it very hard for the author to assume and use a consistent vocabulary in writing the rest of the manual. Leaving out the second leaves the reader without any sense of what is important and how to use the CBS to solve his problems.

A well-written second part makes reading the manual, even the third part, fun. The third part must be consulted in order to fully explain why the input of an example solved the problem the example claims it does.

A good way to organize the first part is around the abstractions that are found in the problem domain. Each abstraction that survives the analysis should be explained in terms of

1. what the objects are,
2. what they do, and
3. what is done to them.

A good way to organize the second part is around the use cases that have been identified in consultation with the client and the users. One can follow any approach to identify use cases [25, 49, 8]. Having identified them, it is helpful to decompose them into two groups:

1. basic use cases that are used frequently as components of other use cases, e.g., selection of text in a WIMP interface, and
2. more complex, problem-solving use cases, e.g., changing the size and position of a selected box in a picture drawing program.

The second group can be sorted into a list by increasing complexity. This sorted list can be used as the basis for the graduated set of examples around which to write the user's manual.

Indeed John and Dörr [26] describe an approach for eliciting requirements for an enhancement of legacy software based on existing user documentation, sort of going in the reverse of our direction. Also they have observed the strong connection between a well-written user's manual and use cases. Namely, the table of contents of such a user's manual is a list of use cases for the legacy system.

The third part is generally a feature list, often in alphabetical order by the feature name. The organization and contents of this part are similar to those of a traditional feature-centered SRS. Thus, those, e.g. designers and implementers, who prefer a feature-centered RS get it also.

According to Fairley in his 1985 *Software Engineering Concepts* [17], a preliminary user's manual should be produced at requirements definition time to get a focused user's view. He proposes the following outline for the preliminary as well as the actual manual:

1. Introduction
   - Product overview and rationale
   - Terminology and basic features
   - Summary of display and report formats
   - Outline of the manual
2. Getting started
   - Sign-on
   - Help mode
   - Sample run
3. Modes of operation:
   - Commands
   - Dialogues
   - Reports
4. Advanced features
5. Command syntax and system options

Observe that Chaps. 2 and 3, about Getting Started and Modes of Operation, are precisely a list of scenarios, dressed up as a list of problems with which the user might be faced, and how to solve them with the CBS being described. The chapter about Getting Started can show the entire scenario of a sign-on to the application and a use of the application to solve a single representative problem. It can then give the basic use cases that are used in other use cases that are the subject of the chapter about Modes of Operation. This latter chapter can consist of the graduated set of use cases that involve using most, if not all, features.

5 CBSs admitting user's manuals as RSs

The approach of offering a user's manual as the principal or only RS of a CBS works only for those CBSs for which a user's manual describes all but trivially explained requirements. Thus, the CBS being specified must have the following properties:

1. The CBS must have at least one kind of user.
2. The CBS must provide all but trivially explained functionality through at least one user interface, and

all of this functionality must be well understood from descriptions of the behavior seen by a user.

3. Each of the CBS's nonfunctional requirements must be well understood and easily described in prose.

If a CBS has several kinds of users, one user's manual can be written for each kind. However, then achieving and maintaining consistency of all user's manuals becomes a problem.

Examples of excluded CBSs are:

1. Autonomous systems with no real human users: However, if a user of the autonomous CBS is another CBS, a manual directed at the users of the other CBS might be able to describe all the functionality assumed of the autonomous CBS. If the autonomous system reacts to real-world phenomena, e.g., temperature changes, a user's manual written from the perspective of the world, e.g., the weather, as a user might be possible.
2. A CBS for which one or more algorithms it computes is the major issue of an RS: An example of such a CBS is a weather predictor, whose core functionality is a particularly effective weather simulator, and the algorithm of the simulator is the issue of the RS. Other examples are programs to approximate the integral of functions and programs to do life-like animations.
3. A CBS with significant, nontrivial nonfunctional requirements that are not specifically visible to the user, e.g., security, reliability, and robustness, for which the user does nothing to cause the nonfunctional requirement to kick in: The particular way that security, reliability, or robustness is achieved is a major issue for the requirements specification.

For none of these CBSs would a user's manual serve as a good RS.

These limitations notwithstanding, the approach of taking the user's manual of a CBS as the RS of the CBS is still helpful, particularly when the alternative is that no RS would be produced.

It should be clear that the limitations of user's manuals with respect to covering requirements are the same as the limitations of scenarios and use cases [49]. They can cover only those requirements that are visible to users. However, users of scenarios and use cases are quick to point out that even a complete collection of scenarios and use cases do not constitute a RS.

We have found an approach for dealing with the well-understood, easily described, common nonfunctional requirements typical of user-centered CBSs. These include such nonfunctional requirements as ease of learning, ease of use, and performance. These can be specified in what we call the "bragging" section that many user's manuals have. A typical such user's manual congratulates the user for having bought a great product with a great user interface and high performance. If the sole specification of a nonfunctional requirement is simply its name, this name could be mentioned in the bragging section. If the specifications are more detailed, then other possibilities exist:

- stating what functions provide the easy-to-learn or easy-to-use interface in the equivalent of a Q-FD deployment of qualities to functions, or
- giving actual performance data, e.g., "If you are the only user of the system on which our product is running, if you are running on a CPU with at least a 50-GHz cycle, all commands but ... will respond in less than one second." or "Our product is capable of editing files of size up to 10 MB."

These details may be appropriate for the bragging section or for the system requirements section.

Even if some nonfunctional requirements of a CBS are not appropriate for specification in a user's manual, it is still useful to specify the CBS's functional requirements with a user's manual, especially if the CBS is user centered. Should it be necessary to specify nonfunctional requirements, then a separate, companion SRS can be written for just the nonfunctional requirements. In any case, no matter where the nonfunctional requirements are specified, the functional information in the user's manual, that is the use cases, may be usable to help identify efficiency and other nonfunctional requirements [14].

6 Case studies

This section describes case studies by the authors using the user's manual for three different CBSs as the RSs for those CBSs. The first case study, for the development of `flo` is by Berry, based on his experience supervising a master's thesis years earlier. The second case study, for the development of `WD-pic`, is by Berry, based on his experience supervising another master's thesis years earlier; by Berry, Daudjee, Dong, and the Nelsons, based on their experiences in a graduate requirements engineering seminar; and by Ou, based on her experience doing her master's thesis under Berry's supervision. The third case study, for the development of `ExpressPath`, is by Fainchtein, based on his experience writing a user's manual as an RS for an industrial product as part of the research for his master's thesis under Berry's supervision.

6.1 `flo`

In 1988, Berry assisted in the writing of a user's manual as a primary requirements document. This experience had an interesting twist to it that cannot happen for all software; however, it is interesting from the total software engineering perspective. The program was `flo` [51], which is a `pic` [30] preprocessor, which in turn is a `ditroff` (device independent `troff`) [39, 31] preprocessor. `flo`'s purpose is to translate a flowchart specification, which is embedded inside a file containing `ditroff` input, into a `pic` specification, which in turn is

translated by the pic program into more ditroff input. The flowchart specification is a textual algorithm in a Pascal-like notation. The generated pic specification describes the flowchart as a picture, and the generated ditroff input draws the individual shapes of the picture. The specified flowchart is constrained to be no larger than a single page, because the output of the ditroff program is a sequence of programs in a page-description language such as PostScript [1]. Each such program prints a single page of the typeset document.

There were two roles in the development of flo, the client and the software engineer (SE). Berry, the advisor, was the client, an occasional writer of computer science theory papers, representing all people who use flowcharts in formal papers. Berry even occasionally asked a theoretician at the Technion, Nissim Francez, for his opinion on features. Wolfman, the master's student, was the SE. He was designing and implementing flo as his master's thesis research under Berry's supervision.

To design and implement flo, Wolfman, with Berry's feedback, wrote an initial user's manual to describe all features that would be implemented. They iterated on the initial user's manual until they were satisfied that the specified flo could be used to draw all flowcharts that they had seen in recent books and papers on theory of computing and software engineering. That is, the user's manual became the RS. This first version of the user's manual used hand-coded pic descriptions to draw each flowchart in the manual to look as if flo had done it. These hand-coded pic descriptions were also Wolfman's idea of the kind of output that flo would create for the flo input.

The following steps were carried out simultaneously and repeatedly:

– Wolfman implemented features in flo and commented out hand-coded examples in the manual source in order to let the flo-generated pic code show through.
– Wolfman, with Berry's feedback, modified the manual to reflect changes in flo's implementation that were necessitated by discoveries made during failures to implement features as desired and as described in the manual.
– Wolfman modified the implementation to reflect changes in the manual that were indicated by discoveries made during attempted use of the features described in the manual to write a changed manual.

The three steps were repeated until they arrived at a manual and an implementation for which

– all implemented features were described in the manual,
– all features described in the manual were implemented, and
– the customer and user, Berry, was happy with the features and their output.

As a bonus, there was available at all times during the implementation a nice built-in test with full feature coverage, the manual itself!

From the similarity in the structures of the papers and manuals about eqn, pic, grap, dag, and flo, it *appears*[3] that the same approach was used by Kernighan, Bently, Cherry, and Trickey to design and implement eqn, pic, grap, and dag!

During the development, the manual underwent many, many iterations.

– Any time Berry did not understand what it was saying, he complained.
– Any time he could not see the purpose of something, he complained.
– Many times, something it said suggested to him another option.
– Many times, something it said led to his asking how to do something related.

Wolfman had to fix each of these problems, sometimes by changing or extending the language (and almost never reducing the language!). In one case, he threw out a whole collection of attributes, "short", "tall", etc., in favor of setting the size of bubbles around nodes; bubbles turned out to be a simple way to specify completely the compactness of the layout. The iteration continued until Berry could think of nothing wrong and nothing more to add. For more details, please consult Wolfman's thesis [50] or a paper derived from it [51].

flo's development followed a waterfall lifecycle. Unlike the traditional waterfall, this one was centered on the production of the user's manual. The feedback loops always led into the requirements specification box (labeled "SPECIFY ..."). Thus, the manual was worked on in all steps (Fig. 1).

6.2 WD-pic

WD-pic (WYSIWYG Direct-manipulation pic) is a WYSIWYG direct-manipulation drawing program whose internal representation is the pic language, the

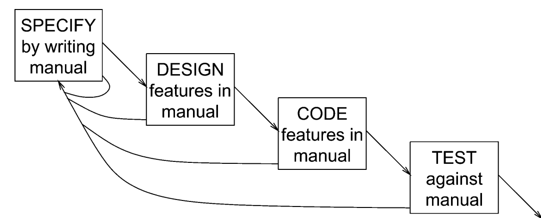

Fig. 1

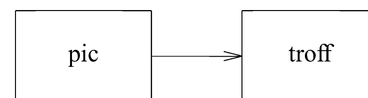

Fig. 2

[3]Appearances can be deceiving. Berry once gave a talk about flo to an audience that included Brian Kernighan, an author of pic. Berry asked Kernighan if this approach was used to design and implement pic. Kernighan replied, "No".

picture drawing language, whose processor is a preprocessor in the `ditroff` family.

The `pic` language allows written descriptions of line-drawn pictures that are typical in computer-science literature. For example, the `pic` specification

```
.PS
box ''pic''
arrow
box ''troff''
.PE
```

gets translated into `ditroff` instructions that cause `ditroff` to display the text shown in Fig. 2. A key point about `pic` is that every graphical object has a default size, e.g., 0.75 in.×0.5 in. for a box, and default position, to the right of the previous item. Thus, many times, the user does not have to provide explicit size and position specifications.

In most WYSIWYG direct-manipulation drawing systems, after one selects a graphical item in the palette, one has to move the mouse over to the canvas in order to specify position and size by clicking twice, once at each corner, or clicking at one corner and dragging to and unclicking at the other corner. In `WD-pic`, if everything is of default size and position, one should be able to draw a whole picture without ever moving the mouse from the palette. To get Fig. 2, one could

– click the `box` button,
– click the `quote` button,
– type "`p`", "`i`", "`c`", "`"`",
– click the `arrow` button,
– click the `box` button,
– click the `quote` button, and
– type "`t`", "`r`", "`o`", "`f`", "`f`", "`"`".

without ever having to move the mouse out of the palette or even to open up a text window. The grammar would tell `WD-pic` that it should expect characters to be entered at the keyboard after the `quote` button is clicked.

Of course, in `pic`, the user may further qualify a graphic item with size and positioning information. Therefore, in `WD-pic`, the user should be able to provide additional size and position information, by either

– clicking some other buttons on the palette,
– editing the internal representation, or
– dragging graphical items on the canvas to the desired size and position.

The first version of `WD-pic` was designed and implemented by Shpilberg, one of Berry's master's students at the Technion. Shpilberg built her version of `WD-pic` from requirements up with Berry as her customer. The main goal of the thesis was to establish the requirements of `WD-pic` by an extended prototyping process, similar to that for `flo`, in which the user's manual served as the RS [44].

While the prototyping did establish the basic requirements and did prove the concept, Berry was less than happy with the final prototype because a number of features were not quite as desired. That they were not right was no fault of Shpilberg. Rather, the problems were the result of her use of a standard graphical user interface (GUI) builder that provides a more cumbersome user interface than desired. The standard GUI builder, used to make prototyping faster, demands clicking an `OK` button to confirm the correctness of text entered into a text window. That such confirmation is unnecessary is indicated by the fact that a user almost never clicks the other non-OK button, and in the rare case that he would, the undo facility could be used to cancel the input. Berry vowed that the production version would have its own lighter-weight GUI that took advantage of the program's knowledge of `pic`'s grammar to know when text begins and ends, thus making the use of a text window totally unnecessary.

After Shpilberg finished her thesis, Berry thought it was time to try to get an improved version going. Consequently, in a graduate seminar on requirements engineering taught by Berry at the University of Waterloo, four of the students that are the other authors of this paper carried out a project on improving the first version of the `WD-pic` manual. They produced three different manuals, two by one-person teams, Daudjee and Dong, and one by a two-person team, the Nelsons. The objective of the project was to determine if the user interface (UI) of `WD-pic` could be redesigned to improve user interaction with the features of the CBS.

The RSs produced by Dong, Daudjee, and the Nelsons were never implemented. These RSs were produced as the term project of a single-semester requirements engineering seminar, and there was no follow-on course to implement the specified CBSs.

About a year later, Ou approached Berry about doing a master's thesis under his supervision. She chose, from among several possible projects, the RS and implementation of the first production-quality `WD-pic` for the customer Berry. Ou was to write the RS of `WD-pic` in the form of a user's manual. This user's manual was

1. to describe all features as desired by the customer, and
2. to be accepted as complete by the customer.

before any work on the design or implementation was started. Once implementation started, when (not if) new requirements were discovered, the manual should be modified to capture the new requirements, so that in the end, the manual described the program as delivered.

The customer was determined to have a specially handcrafted GUI taking advantage of the program's knowledge of the current state of parsing its input from the user so that there would be no need for confirming with a mouse click the acceptability of what he just finished typing at the keyboard.

It should be mentioned that prior to entering graduate school, Ou had built other systems in industrial jobs, mainly in commerce. In these jobs, she had followed the traditional waterfall model, with its traditional heavy weight SRS, and had made effective use of libraries to simplify development of applications.

After accepting the project as her thesis topic, Ou wrote a project plan that reflected her waterfall model experience (Table 1). The first month's preparation consisted in learning `pic` and studying and playing with Shpilberg's prototype, and studying the RSs written by Daudjee, Dong, and the Nelsons. Based on this information and input from Berry, Ou was to scope the project to a coherent set of requirements for a first production-quality version of `WD-pic`.

The actual project history was a bit different, with Ou spending a lot more time than planned in requirements specification (Table 2).

While the detailed plan was not followed, the total project time was as planned. Moreover, Ou ended up producing two implementations for the price of one, the planned one for the Sun with UNIX and another for the PC with MS Windows 2000. That Ou finished on time was more of a surprise to her than to her advisor Berry, who had a lot of faith in the power of good requirements engineering to reduce implementation effort. Adding to Ou's surprise was that the requirements phase took nearly 5 months instead of the planned 2 months. In her view, the schedule had slipped 3 months out of 10, way beyond recovery.

In Ou's past experience, as reflected by her long projected implementation and testing times and the 1-month buffer, implementation is normally slowed by the discovery of new requirements that necessitate major rewriting and restructuring. This time, however, thanks to the completeness of the RS, there were only minor rewriting and no restructuring. Thus, instead of spending 2 months specifying and then 7 months implementing and testing, she spent 5 months specifying and then only 4 months implementing and testing.

Table 1

Duration (months)	Step
1	Preparation
2	Requirements specification
4	Implementation
2	Testing
1	Buffer (probably more implementation and testing)
10	Total planned

Table 2

Duration (months)	Step
1	Preparation
4.9	Writing of user's manual ≡ requirements specification, 11 versions
0.7	Design, including planning implementation strategy for maximum reuse of pic code and JAVA library
1.7	Implementation, including module testing and three manual revisions
1.7	Integration testing, including one manual revision and implementation changes
10	Total actual

By spending 3 months more than planned writing an RS that satisfied a particularly hard-nosed customer who insisted that the manual convince him that the product already existed, Ou managed to produce an RS that had very few errors and that was very straightforwardly implemented. That is, there were *no* show-stopping surprises that would cause a major restructuring or rewriting. It helped that Berry, the customer, knew pretty much what he wanted and did not want from his experience with 1 prototype and 3 other RSs. Almost all the errors found by testing were relatively minor, easy-to-fix implementation errors. The two requirement errors were relatively low level and detailed and involved subfeatures in a way that required only very local changes to both the manual and the code.

Particularly helpful was that all the exceptional and variant cases had been worked out and described in the manual. Thus, there was very little of the traditional implementation-time fleshing out of exceptional and variant cases and the traditional implementation-time subconscious requirements engineering.

Consequently, Ou found the scenarios, including exceptions and variants, that were in the manual to be a complete set of black box test cases, so much so that, much to her surprise, scenarios not described in the manual, but which were nevertheless logical extensions and combinations of those in the manual, worked the first time! That is, the features composed orthogonally without a hitch.

Berry found Ou's implementation to be production quality and is happily using it in his own work. More details on Ou's work can be found in her master's thesis [40].

6.3 `ExpressPath`, an industrial case study

The industrial experience involves Fainchtein, one of Berry's master's degree students, writing a user's manual as an RS for `ExpressPath`, a natural language speech recognition system developed by the LGS Group. `ExpressPath` uses speech recognition to allow software to answer telephones and to satisfy users' requests in a voice interface. This voice-only interface is more convenient to the user than the traditional labyrinthian menu-driven interactive voice response (IVR) system that uses the telephone's touch-tone keyboard for input and that has become the much maligned standard for possibly toll-free telephone numbers provided by merchants, services, and governments.

Note that as with `WD-pic`'s user interface, `ExpressPath`'s voice user interface (VUI) is not well-known by the user community. Therefore, a significant part of the requirements engineering (RE) effort is the design and structure of the user interface. There are no body of experience, guidelines, and standards upon which to draw. `ExpressPath` will be setting the standards. Consequently, difficulty understanding the user's manual raises a concern regarding the usability of the described system.

Full details on `ExpressPath` and the experience specifying it with a user's manual are found in Fainchtein's

master's thesis [16]. Fainchtein was a member of the team developing ExpressPath while he was studying for a master's degree in a special program that allowed studying for the degree while still working. Fainchtein had just taken Berry's Software Requirements and Specification course for the degree program. In this incarnation of the course, Berry had assigned as the term project the production of a WD-pic specification in the form of a user's manual. Fainchtein was searching for an advisor and topic for his master's thesis. He approached Berry with the proposal of applying the idea of writing a user's manual as the RS at his job, in which he was a developer in the project to build ExpressPath. Berry pounced on the idea as a way to get a much-needed industrial case study of the proposal of using a user's manual as an RS. However, Berry was worried that Fainchtein's managers might not be happy about their project, with real market-driven deadlines, being used as a subject in academic research about a technique that is not well known.

Fortunately, the project was in the early stages. In previous projects at the company, all in the domain of IVR, the company had used RSs in the form of use cases agreed to by the customers and the developers. Each such use case was described in words and with flowcharts, showing the use case's main, alternative, and exceptional scenarios. Each scenario is a tree of menus and choices with voice prompting by the application and telephone keyboard (DTMF) input by the user. These use cases sufficed as RSs because the applications were in a well-understood domain, IVR, with which users were intimately familiar. Over the years, principles about IVR user interfaces had been developed and refined to the point that, for any given application, it was easy to gain a customer's understanding of any proposed solution without the need to consult each category of users directly.

This new project with a VUI was in a totally new domain for which there was *no* user experience. In order to gain some experience during the development, the company decided to deploy its developing product experimentally on its own phone system. Partly because of market pressures to get the application out early and partly because of the normal use of tacit assumptions, the company was following its traditional requirements process, producing use cases as RSs. Each such use case was again a tree of menus and choices with voice prompting by the application and voice input by the user. In other words, they were basically the same IVR application use cases with a change of input medium from keyboard to voice. These use cases were being accepted without reconsidering the tacit assumptions and without consulting any category of users except the developers, who were the users of the in-house experimental deployment. Occasionally these user–developers were prototyping particularly troublesome scenarios, but again without consulting any category of users but themselves.

Fainchtein realized since the VUI was entirely new technology, less familiar to the customers and developers, it was necessary to revise the requirements process. It was necessary to institute a requirements process

1. involving all categories of the users, and much more often,
2. rethinking the application from the ground up,
3. questioning tacit assumptions,
4. exploring entirely different alternatives, and
5. producing an RS that described the application thoroughly.

The goal of describing the application thoroughly would force consideration of all the details and questioning of tacit assumptions. Fainchtein believed that writing a user's manual as an RS would be a good method to do what was needed and to get the desired RS.

Fainchtein approached the project leader with the idea to produce an RS in the form of a user's manual. This idea did not appeal to the manager since he had never taken this approach before, and therefore, he perceived the technical risk to be too great. However, he understood that

1. some RS would be useful even though he did not perceive that he had the resources for it or that it was valuable enough to commit his precious resources to it, and
2. eventually, a user's manual would need to be written.

Fainchtein offered to write the user's manual on his own time—the company was already supporting his studies for the master's degree and had already set aside some of his time for his studies. The project leader agreed that Fainchtein would write a user's manual on a part-time basis during time allotted to his studies, but he could interact with project personnel and other stakeholders freely. Thus, the manager got a needed document at a cost reduced to well below the document's perceived necessity, at a considerably reduced technical risk, and with a very high potential payoff in reduced costs, errors, etc.

Note that the user's manual was being written while rapid prototyping was being used to identify requirements. Thus, the user's manual was not functioning as a poor man's prototype; it was serving merely as an RS. However, the user's manual did make prototyping of some ideas unnecessary. Once an initial prototype had established a particular paradigm in users' minds, variations described in the form of scenarios in trial manual sections were clear and concrete enough to allow selection of the best variation without having to actually implement them in a prototype for presentation to the users for trial.

Production of the user's manual as an RS required 5.5 months of part-time work by Fainchtein, in the midst of full-time work on and thinking about the system. The company's experience with similar sized projects suggests that normally about 5 months[4] would be needed for writing *each* of an SRS and a user's manual. Hence, Fainchtein's work represents a significant savings

[4] In each case an actual metric is compared with an expected metric, the expected metric is based on the company's past history with projects of similar size and complexity. There is no way of knowing exactly what the other way would have cost short of doing the project twice—and no one is going to do that—with carefully selected skill-balanced groups.

overall. They saved time by writing only one document instead of two. Moreover, all involved believe that the implementation is going more rapidly and error free than normally, and they believe that this more rapid development is happening because requirements were clarified and documented in advance of the implementation. Thus, Fainchtein has gotten one document serving as two for the price of one, and he has gotten it early enough to more than pay in reduced development costs for the time he spent in writing the manual.

During his work on the manual, Fainchtein got the usual number of complaints with regard to specified requirements. However each of these complaints resulted in a new iteration of the manual and resolution of the problem during requirements analysis time rather than during testing or after delivery. Moreover, having the RS in the form of a user's manual rather than in the traditional SRS form made it easier for the customer to understand the requirements. Consequently, the customer's complaints were more specific and more constructive, and they often contained a solution. In other words, the customer knew very well what he wanted. The customer's knowledge of the requirements can be attributed partially to the readability of the user's manual and partially to the accessibility and concreteness of the prototype. Team members other than Fainchtein found it easier than normal to elicit customer feedback using the user's manual.

Fainchtein's participation in the entire requirements analysis and system design, including construction of the prototype, helped him to produce the user's manual in what felt like a short time period. Implementation of some advanced functions and of a major portion of the graphical user interface in the prototype allowed Fainchtein to include screen shots in the manual, thus increasing the manual's realism. The resulting impression that the system already existed offset some of the understandable difficulties that arose from the fact that the manual was written in advance of the availability of a complete implementation.

Each portion of the user's manual, before being presented to the client, had undergone much discussion and modification by the requirements analysis group, consisting of Fainchtein, another solution developer, a solution architect, and a domain expert. They found that the processes emulated by the prototype system were easier to describe in the user's manual than in the more formal SRS that they eventually wrote. The customer also validated these user's manual descriptions faster because he was able to see how certain functions were being used by a user. The customer went so far as to say that presenting the RS to him in the form of a user's manual as opposed to a traditional SRS contributed to a reduction in the overall time spent by his representatives to validate the requirements and allowed him to involve more domain experts and those who had no previous experience dealing with a traditional heavy SRS.

As confirmed by the originally skeptical project manager, prototyping and the user's manual allowed requirements analysis group to detect in the *first* iteration more than 75% of all problems that they have found in the implementation. These problems were fixed immediately, and the fixes were reflected in the next versions of the prototype and the user's manual. More generally, the manager observed that the manual was a source of test cases for the quality assurance personnel. He noted also that the customer expressed satisfaction with the fact that the requirements processes allowed the customer and the developers to detect and readily address any human–computer interaction problem that arose during requirements specification.

The analysis team found that having an RS in the form of a user's manual made it easier than normal to work with the customer to address potential human–computer interface issues. Normally, a problem with this interface would not even be detected until after the system were implemented. This benefit is important in an application for which there is little prior experience with its user interface, as is the case with VUI.

The developers find the user's manual easy to work with as an RS. Some of them are new to the group. The user's manual form of the RS has made it easier for the new members to get up to speed. Having both the user's manual and the prototype has reduced the learning curve by at least 50% over having only a traditional SRS. Indeed, the developers have gotten into the habit of calling the user's manual "the specification".

7 Lessons learned

In the course of the experiences described above and others, a number of lessons were learned. They may be classified into five different groups:

1. advice on writing user's manuals in general and as RSs,
2. special kinds of user's manuals,
3. why user's manuals work as RSs,
4. requirements engineering processes aided by writing a user's manual, and
5. other software engineering processes aided by writing a user's manual.

This section is divided into five subsections, one for each group of lessons.

7.1 Advice on writing user's manuals

We learned some specific lessons about writing user's manuals, both in general and for user's manuals that function as an RS.

7.1.1 User's manual should deceive users

The manual should be written well enough that it deceives the reader into believing that the software really exists. In fact, it's getting the picky details worked out well enough to write the deceptive user's manual that forces ironing out those synergistic problems that plague many requirements, and even design, documents. We have here yet another example of faking it [41].

7.1.2 *Manuals should be written in present tense*

An important rule for writing user's manual, and for that matter any RS, is to use present tense to talk about what the CBS *does* (notice the present tense in this sentence!). This rule is certainly consistent with the idea of faking it that the CBS is already implemented when writing the user's manual. This rule is needed so that when it is necessary to talk about something that happens in the user's future after the user's present in which he says something to the CBS, future tense can be used to distinguish the future response from the user's present input. A typical specifier writes an RS for a not-yet-implemented product in the future tense. After all, after the CBS is implemented in the future, the user will enter some input, and the CBS will do something in response. When the RS is written in future tense, the specifier has lost the ability to distinguish between the user's current time and the user's future. Everything happens in the specifier's future.

7.1.3 *When "how" information is needed in specification*

Recall that we are admonished to specify what, not how when giving an RS for a CBS. Specifying only what allows the implementers the greatest freedom to choose an implementation, a how. However, several have noted that sometimes it is necessary to give some details of the how, usually what is now termed "architecture" [47, 5, 38, 37]. One example is that of Knuth's exposure of the line-breaking algorithm for $\mathrm{T_EX}$ in *The $T_EX$book* [32], which serves as both the RS and the user's manual. Knuth described the algorithm in the RS-and-user's manual for two main reasons:

1. to ensure that *all* implementations of $\mathrm{T_EX}$ produce the same formatted output, even down to the line breaks and spacing between words and
2. to give the user enough smarts about the line-breaking algorithm that he can exercise the commands to effectively achieve the desired line breaking and interword spacing.That the RS of $\mathrm{T_EX}$ is its user's manual seems to have served as a powerful filter to make sure that a how detail showed up in the manual only when the detail is necessary for the user's effective use of $\mathrm{T_EX}$.

7.1.4 *Prototyping helps write user's manual*

Sometimes, in order to write a user's manual, the requirement engineer has to prototype or at least mock up the screens so that she can get the nice screen pictures to put in the manual. This prototype helps also to get the client to validate the requirements.

7.1.5 *Skills needed to write a good user's manual*

Writing a good user's manual takes skill, and there is no substitute for that skill. In an industrial situation, the client and the CBS producer must hire good writers to write good conception and requirements documents. The skills to write a good RS include general writing skills. These skills may be more important than that of understanding the CBS to be built. Glass reports how non-software-knowledgeable English majors were successful in writing high-quality descriptions of the grubby details of programs in documentation about these programs for maintainers [19].

7.2 Special kinds of user's manuals

Sometimes user's manuals are written for users of a system other than the ordinary user of an application. An example, mentioned in Sect. 2.2, is the system maintainer user. There are yet others. Also, a help system should be an acceptable substitute for a user's manual as an RS.

7.2.1 *Requirements and user's manuals for platforms*

A computing platform, such as an operating system, is one kind of software that has a special class of users that differs considerably from the class of users for a single application. The user of such a platform tends to be a sophisticated user who programs applications for use by others. These applications are programmed to run on the platform. The equivalence of RS and user's manuals holds even for these platforms. Consider the POSIX system [42], which is a standard generalization of the various UNIX platforms. It is specified by a collection of UNIX-style manual pages describing the various kernel routines and data that are available to use to write applications running on the platform.

1. This collection of manual pages serves as a description of the facilities that an implementation must make available. This description gives for each kernel routine the interface it must support.
2. This collection of manual pages serves also as a user's manual describing what the programmer of an application may assume about the facilities on which the application is programmed. The description gives for each kernel routine the interface that can be assumed by its invoker.Any user of any POSIX-compliant operating system is supposed to be able to rely on the POSIX specification as a user's manual for the particular operating system. Thus, even for an operating system, a well-written user's manual can serve as an RS. Of course, this style of user's manual, aimed at the programmer of applications, may not appear well written to the user of such an application.

7.2.2 *Help systems as user's manuals*

A question that is asked a lot these days when purchasing mass-market software is "Where have all the manuals gone?". Software is rarely delivered with a full-fledged user's manual covering all functionality. Instead, a manual covering only installation and initial start up is provided, and the installed software comes with a *help system*. The help system replaces the full-fledged user's

manual in describing all the implemented functions from the user's point of view. The help system is organized around use cases and scenarios. Indeed, the table of contents of the help system is basically a list of use cases. Each selected page gives one or more scenarios for the use case. A properly constructed help system provides the same information as a properly constructed user's manual. Therefore, we believe that constructing a help system should be just as effective for determining and specifying requirements as writing a user's manual.

7.3 Why user's manuals work as requirements specifications

On several occasions, we put our fingers on explanations for why user's manuals make good RSs.

7.3.1 Exposing intent and prototyping

Our experiences with writing and refining the `WD-pic` user's manual show that the user's manual is a good representation of `WD-pic`'s requirements. The user's manual describes more than the requirements, i.e. it captures the requirements of `WD-pic` while also providing the user with step-by-step instructions of how to use the features of `WD-pic`. This description amounts to a statement of intent [35, 43]. Understanding this intent is critical for the implementers, who may not have access to the specifiers. When the implementers have a question about the meaning of the RS, in the absence of access to the specifiers, knowledge of the intent helps the implementers to figure out the intended meaning. The process of improving the user's manual is iterative and can serve to incorporate new features into the CBS through iterative development. It is not an accident that iterative development of a user's manual parallels the iterative development process of software systems. Since the user's manual allows the user to imagine interacting with the CBS, it serves to provide feedback on whether the CBS incorporates the required features of the CBS. In addition, it also allows the developers to gather new requirements for the CBS. In this context, the user's manual can be considered to be a poor man's prototype, and the writing and improvement of the user's manual is a rapid prototyping process.

This form of rapid prototyping is an inexpensive process. Little or no code has to be written, and no complete, running product has to be developed. At most a few screen mockups have to be written to create illustrations for the manual. It is a poor man's prototype. The resulting user's manual serves as a cost-effective and low-cost framework and knowledge base for the later development.

7.3.2 Scenarios for requirements

Use cases and scenarios are a good way to represent the user's requirements [29]. Because use cases and scenarios are more malleable than design or software, they are more easily discarded than design and software [11]. Indeed, in our case studies, no one really complained about throwing out sections of the user's manual per se. The complaints, if any, were "When are we going to finish this endless cycle of revisions? Can't the d—n customer make up his mind?". However, the same complaint would be raised if code were changed as often in order to be able to match the customer's requirements exactly. The alternative is to quit early and never match the customer's requirements.

In fact, each of the user's manual authors among this group of paper authors found that throwing out portions of the manual that he or she had written was far less distasteful than throwing out portions of code that he or she had written. After all, the manual portion is only words, but code is code! Perhaps this is a phenomenon restricted to programmers, as opposed to literature authors, but sentences from a manual seem a lot less part of one's person than do lines of code. More on the rational side, a manual section seems a lot less connected to the rest of the manual than a code section is connected to the rest of the code, so it's a lot less disruptive to throw out a manual section than to throw out a code section.

7.3.3 Ease of writing specification as a user's manual

The Nelsons observed that they did not find the process of writing the RS for `WD-pic` difficult, because they were not really concentrating on writing a traditional SRS document. Instead, they were writing a user's manual, which is a familiar document with a well-known structure. Even more important, it is a user-centered document that promotes the requirements elicitation process. For instance, they frequently found themselves asking questions like "So, if the user wants to move an object, can he find out how to do it by reading the manual?" or "How will the system respond if the user tries to select a group of objects?".

7.3.4 For designers and implementers

It can be argued that a user's manual favors the user over the designer and implementer. The audience of the user's manual is the user. Certainly with a user's manual being use-case centered, it will be hard for the designer and implementer to identify functions to be implemented. A given function may manifest itself in several use cases, each giving some different aspects of the functions. With only this information, the designer or implementer would have to distill the functions out of their many manifestations.

However, a good user's manual has also a feature-centered part listing all the individual features and describing all of the options of each. Indeed, this part is organized much as is the typical feature-centered traditional SRS. The designer and implementer can find the functions already distilled out in this part.

7.3.5 User's manuals engage users more than SRSs

It is our experience that a user's manual engages the users and customers much more than any more formal

requirements specification, including the traditional SRS. A user's manual is written addressed to the user and generally talks to the user. An SRS is written addressed to the developer and generally fails to talk to the user.

7.4 User's manuals and requirements engineering

We found that writing a user's manual forced us to do some RE processes that might otherwise not have been done.

7.4.1 Writing user's manual as an elicitation and validation tool

Giving a draft user's manual to the client and to the client's users is a good way to elicit the "But, I wanted *x* instead"s and the "But, I wanted *x* also"s before the CBS is implemented, i.e., to validate the requirements! The user's manual actually serves a dual purpose: aiding the elicitation of correct and complete requirements, and being the requirements document.

7.4.2 Finding ambiguities

In doing the `WD-pic` project, we found that one of the most important benefits of using a user's manual as an RS is that it is easier to find ambiguous RSs. In any RS document, there may be hidden ambiguities. Each such hidden ambiguity gets subconsciously disambiguated one way by each stakeholder, e.g., the user and the implementer, who is not even aware that other meanings exist for the sentence [28]. The ambiguities are discovered only later when the CBS is delivered and the user discovers that the implementer did not implement what she understood of the RS. In our experience in the production of several user's manual specifications of `WD-pic`, the students clarified a number of misunderstandings with Berry by showing him a UI and a number of use cases. His reactions to the UI and the use cases made the ambiguities and his intent clear and helped to resolve the ambiguities.

7.4.3 Finding unexpected implications

Very often, unexpected interactions between requirements are not found until the code is written or even until the CBS is deployed. We have found the process of writing and validating a manual effective in helping the requirements engineer to find these interactions earlier than they might otherwise be, particularly if scenarios exercising the requirements are described in the manual. An example is the use of a grid in `WD-pic`. When the Nelsons tried to explain to the user how to use a grid to position objects, they noticed that enabling a grid would affect the behavior of other features in the system. Although the grid was one of the initial requirements, the interaction with other features and subsequent effect on their behavior was noticed only when writing the user's manual. The Nelsons then had to study the other features and revise their manual to reflect the feature interactions. Daudjee experienced the same phenomenon in the writing of his user's manual.

7.4.4 Details tradeoff and no handwaving

No requirements specification method that does not force working out the details is going to work. It is only in working out the details that all the show-stopping exceptions and interactions are going to be discovered. These details can be worked out in any of several media:

- the software itself,
- a complete formal specification,
- a complete traditional SRS, or
- a complete, scenario-based user's manual.

The advantage of the user's manual over the other media is that changing the manual consistently is much cheaper than changing either the software itself or a complete, formal specification. Also, unlike a complete, traditional SRS, a user's manual is both needed and perceived as needed after the software is delivered; thus, the motivation to keep it up to date is higher than that to keep a traditional SRS up to date.

The advantage of the software itself or a complete formal specification is that it is hard to hand-wave over the details, to cheat to leave the impression of completeness when details are missing. If details have been left out, the software will not work, or the formal specification cannot be verified to satisfy requirements. While it is fairly easy to leave details out of a user's manual, since the user's manual is intended to be delivered with the software to help naive users, the incentive is to get those details in.

Thus, it is an issue of finding a right medium for expressing detailed requirements that is both cheap to change, but hard to hand wave one's way to a false impression of completeness.

7.4.5 Validation of requirements specifications

One key advantage of user's manuals over traditional SRSs is in the ease of validating the requirements document with the customer and users. This point was driven home to Berry when he compared use-case-based user's manual specifications of the enhanced `WD-pic` with a feature-centered, traditional SRS that had been written by one team for the same. Even though he was quite familiar with `WD-pic` from having used a previous version, even though he is thoroughly computer literate, he had a hard time understanding some specifications of features in the traditional form. He could not see that what was specified was not quite what he wanted. He had no such problems with any of the user's manual specifications. He was able to spot specifications that did not correspond to his desires instantly and to describe what was wrong and how to fix it, or at least what the misunderstanding was. The clarity of the two specifications was like night and day. In fact, he even empathized

with those customers who report that they understand and accept specifications that they were too embarrassed to admit that they had not understood at all.

When he thought about it, he understood what was the key difference in the two kinds of requirements documents. The normal SRS describes only what the system being specified does. The user's manual describes conversations between the user and the system to achieve the user's goals. Basically, the user's manual was more alive; Berry could see himself being the user described in the manual, and he could thus spot instantly supposed user behavior that did not correspond to what he would do. The normal SRS does not describe the user's inputs and reactions. It describes only the system's behavior in a vacuum from the user. So, Berry had no idea what user behavior was implied. Thus, if the behavior bore any resemblance to what he thought the system would do to some input of his, he was led to believe that the specification was what he wanted.

7.5 User's manuals and software engineering

Writing a user's manual helps phases of software development other than the RE phase.

7.5.1 Scenarios as test cases

Recall that Wolfman and Berry used the `flo–ditroff` source of the `flo` manual as the test case for the `flo` program. Of course, this benefit came from the lucky accident that `flo` is a batch program and they wrote the user's manual for `flo` in the formatting language of the formatting system of which `flo` is an integral part. In a similar fashion, Knuth used the $\mathrm{T_EX}$ source of *The T_EXbook* as the main test case for the $\mathrm{T_EX}$ program. In each case, the manual served as the main test case for the formatting software since it describes all the features that are implemented. Such direct use of the manual as a test case is not possible in today's WYSIWYG formatting systems. However, it is still possible to generate test cases from the usage descriptions found in the user's manual.

More generally, scenarios or use cases can be used to generate test cases. A little thought shows a fundamental equivalence between scenarios and test cases. Both must cover all possible inputs and uses of the CBS under consideration, i.e., the CBS under design or test. Obtaining this full coverage is hard. The literature on testing abounds with proof of this difficulty [3, 20, 22].

Indeed, after `flo` had been used about a year with no problems, an input was given that `flo` drew incorrectly, collapsing two boxes into one, with their interior texts overwriting each other. Without going into too many details, the input involved nested conditionals. It turned out that Wolfman and Berry had *never* tested that particular input or any of the infinitely others that showed the problem. There were no such examples in the manual! In retrospect, it is hard to imagine not having nested conditionals as a scenario or as a test case if the goal is complete coverage. However, neither of them ever thought of it. Perhaps they did not view this as a special case, because we had tried other forms of nesting. Moreover, it is so common that no one else thinks of it as very special. The bug showed up one day when they were using `flo` for a production job[5]. Completeness of scenarios and completeness of test cases are tough to achieve, very tough.

There is an added benefit of deriving test cases from the scenarios in the user's manual. Users are strongly influenced by the user's manual and tend to adopt the modus operandi implicit in the way the scenarios choose to solve problems. To the extent that the scenarios are representative of the ways that the users will use the CBS, scenario-generated test cases will tend to cover the probable usage patterns better than arbitrarily generated test cases.

8 Conclusions

Writing a good RS is hard, and it is hard to motivate people to write one. A user's manual is an ideal RS in many cases because if it is well-written

- it is written at the level of what the user sees,
- it describes the basic concepts, and
- it does not describe implementation details.

That is, it is written at the right level of abstraction for an RS.

Certainly, all the case studies have demonstrated that for appropriate CBSs, the production of a user's manual and the user's manual itself serve the purposes of and provide the benefits of producing an RS and the RS itself. However, the issue of whether producing a user's manual helps mitigate the inhibitions against producing an RS remains.

In the academic case studies, there were few inhibitions to producing an RS, even in the form of a user's manual because the customer demanded it and the developers knew that they would not get their master's degrees unless they met their customer's demands. In more than one case study, the student more than once expressed thoughts that the demanded user's manual was overkill, that she was tired of the customer's hard-nosed demands that the manual be perfect, and that the time was long overdue to move on to coding.

In the industrial case study, there was not going to be a suitable RS, and there was not even going to be a totally suitable requirements engineering process. Admittedly, the project leader was not motivated at first to even try producing a user's manual as the RS. It was only when he got an offer that could not be turned down, of a user's manual written on an employee's own time with minimal resource demands on employees working for him, that he agreed to an attempt to produce a user's manual as the RS for his project. Adding to the sweetness of the deal was the realization that a user's manual would eventually have to be written anyway. Even in accepting the offer, perhaps as

[5] Isn't that how *all* bugs show up?!?

a favor to his employee's education that he was already supporting, he remained skeptical as to the effectiveness of the user's manual as an RS. However, once it became apparent to the leader and to the rest of the project team that writing the user's manual was working as predicted as a requirements engineering process and that the were going to get both an acceptable RS and a user's manual for the price of one, the entire project team, including the leader, bought into the user's manual as "the spec".

With enough demonstrations of the effectiveness and cost benefits of the approach, perhaps more projects can be persuaded to adopt this approach as an effective compromise between the extremes of having no requirements process at all and having a full-fledged heavy-weight requirements process producing both an SRS and related documents and a user's manual. While industry is wary of jumping on yet another methodological bandwagon (YAMB), it does adopt techniques that are repeatedly demonstrated in practice to be effective, such as inspection [15]. Perhaps with enough studies like this one showing clear success stories, the user's-manual-as-RS approach will be well adopted.

9 Threats

The reader should remember that these case studies of writing requirements specifications in the form of user's manuals are just that, case studies. Certainly, each case study was a success in that

- the writing of the user's manual helped focus requirements elicitation and analysis,
- the writing of the user's manual forced consideration of user needs in the product and in its requirements,
- the user's manual provides a covering set of test cases,
- in each case in which the product has been delivered, the user's manual describes the product completely,
- in each case in which the product has been delivered, the customer is satisfied with both the end product and the match between the manual and the product,
- in one case, the user's manual helped the programmer meet the overall schedule even though writing it caused a schedule slip in the requirements phase, and
- in the case in which there was not going to be a fully documented requirements specification, the developers have ended up calling the user's manual "the requirements".

However, each of these successes in the non-industrial cases could have been the result of a bunch of other factors, including

- the abilities of the programmer and the client,
- the client's previous experience with the application,
- the user-interface centeredness of the application, and
- the medium size of the application.

These were not factors in the industrial case. In the industrial case, the success could have been the result of

- the Hawthorne effect, since writing a user's manual was a totally new and experimental activity for the people involved, and
- the fact that it was at first not considered a real part of the project, because it was being done as an academic project on someone's own time.

Therefore, it is not correct to conclude that user's manuals will always make good requirements specifications and that writing the user's manual as the requirements specification will always help a project to be successful.

There is no completely satisfactory way to validate any software engineering method. In order to be able to afford to do a statistically significant controlled experiment validating the effectiveness of any method, we have to use many replications of small, toy versions of a problem on which the method is applicable. However, then it is not certain that the conclusions are useful. We know that methods that work on toy problems, of the size needed for a controlled experiment, do not necessarily scale up to industrial-sized problems. Hence, we are left to doing many introspective case studies on applications of the method to industrial-sized examples, and reporting the lessons learned. To be able to measure the success of these case studies, i.e., to show that projects that write user's manuals as requirements specifications finish faster with fewer bugs and a more reliable product, it will help if they were carried out by measuring organizations with track records of successfully predicting resources required for its projects. The organizations could compare the actual project data with past experience and the predictions.

These case studies should be looked at for their lessons learned. They can guide anyone who wishes to try the approach out on his or her own development. If the reader believes that user's manuals make good requirements specifications, he or she should attempt another case study and report what happens.

10 Future work

While there is strong support for the conclusions of this paper, more work is needed to be able to say definitively that for appropriate CBSs, user's manuals make good RSs. We need to continue to track the `ExpressPath` project as it completes development, ships the software, and begins to develop other versions. We will need to report on the complete project history, including milestone adherence, error reports, and even whether enthusiasm for the user's manual as the "spec" is maintained.

Additional case studies, particularly of industrial projects, need to be carried out. The authors would appreciate hearing about any project that has carried out or that wishes to carry out user's manual development as its RS.

Acknowledgements The authors thank the anonymous referees of earlier versions of this paper for incisive, demanding comments. Berry was supported in parts by a University of Waterloo Startup Grant and by NSERC grant NSERC-RGPIN227055-00.

References

1. Adobe Systems Inc (1992) PostScript language reference manual, 2nd edn. Addison Wesley, Reading, MA
2. Anonymous (2003) Is it true that Apple User Manual served as requirements? D. Zowghi (ed) RE-Online e-mail mailing list, 30 January 2003
3. Beizer B (1990) Software testing techniques, 2nd edn. International Thomson Computer Press, London
4. Berry DM, Lawrence B (1998) Requirements engineering. IEEE Software 15:26–29
5. Berry DM (2001) What, not how? When is 'how' really 'what'? and some thoughts on quality requirements. Technical report, Computer Science Department, University of Waterloo, Canada
6. Boehm BW (1981) Software engineering economics. Prentice-Hall, Englewood Cliffs, NJ
7. Boehm BW, Ross R (1989) Theory W software project management: principles and examples. IEEE Trans Software Eng SE-15:902–916
8. Breitman KK (2000) Evolução de cenários (Scenario evolution). Dissertation, Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, Brazil
9. Brooks FP Jr (1975) The mythical man-month: essays on software engineering. Addison Wesley, Reading, MA
10. Carroll J, Rosson MB, Chin G, Koenemann J (1998) Requirements development in scenario-based design. IEEE Trans Software Eng SE-24:1156–1170
11. Carroll JM (2002) Scenarios and design cognition. In: Proc IEEE joint international requirements engineering conference (RE'02), Essen, Germany, IEEE Computer Society, Los Alamitos, CA, pp 3–5
12. Davis AM (1990) Software requirements: analysis and specification. Prentice-Hall, Englewood Cliffs, NJ
13. DeMarco T (1997) The deadline. Dorset House, New York
14. Dörr J, Kerkow D, von Knethen A, Paech B (2003) Eliciting efficiency requirements with use cases. In: Proc ninth international workshop on requirements engineering: foundation for software quality (REFSQ'03), Klagenfurt/Velden, Austria 16–17 June 2003
15. Fagan ME (1986) Advances in software inspections. IEEE Trans Software Eng SE-12:744–751
16. Fainchtein I (2002) Requirements specification for a large-scale telephony-based natural language speech recognition system. Thesis, School of Computer Science, University of Waterloo, Canada
17. Fairley RE (1985) Software engineering concepts. McGraw-Hill, New York
18. Gause DC, Weinberg GM (1989) Exploring requirements: quality before design. Dorset House, New York
19. Glass RL (1993) Can English majors write maintenance documentation?. J Syst Software 21:1–2
20. Gourlay JS (1983) A mathematical framework for the investigation of testing. IEEE Trans Software Eng SE-9:686–709
21. Hooper JW, Hsia P (1982) Scenario-based prototyping for requirements identification. Special Issue on Rapid Prototyping, working papers from the ACM SIGSOFT Rapid Prototyping Workshop. Software Eng Notes 7:5, 88–93
22. Howden WE (1980) Functional program testing. IEEE Trans Software Eng SE-6:162–169
23. IEEE (1998) IEEE recommended practice for software requirements specifications, ANSI/IEEE Standard 830-1998. IEEE Computer Society, Los Alamitos, CA
24. Jackson MA (2001) Problem frames: analysing and structuring software development problems. Addison-Wesley, Harlow, England
25. Jacobson I (1992) Object-oriented software engineering. Addison Wesley, Reading, MA
26. John I, Dörr J (2003) Elicitation of requirements from user documentation. In: Proc ninth international workshop on requirements engineering: foundation for software quality (REFSQ'03), Klagenfurt/Velden, Austria, 16–17 June 2003
27. Kamsties E, Hörmann K, Schlich M (1998) Requirements engineering in small and medium enterprises. Requirements Eng J 3:84–90
28. Kamsties E (2001) Surfacing ambiguity in natural language requirements. Dissertation, Fachbereich Informatik, Universität Kaiserslautern, Germany, also volume 5 of PhD theses in Experimental Software Engineering, Fraunhofer IRB, Stuttgart
29. Kauppinen M, Kujala S, Aaltio T, Lehtola L (2002) Introducing requirements engineering: how to make a cultural change happen in practice. In: Proc IEEE joint international requirements engineering conference (RE'02), Essen, Germany, IEEE Computer Society, Los Alamitos, CA, pp 43–51
30. Kernighan BW (1982) PIC—a language for typesetting graphics. Software Pract Exp 12:1–20
31. Kernighan BW (1982) A typesetter-independent TROFF. Computing science technical report no. 97, Bell Laboratories, Murray Hill, NJ, March 1982
32. Knuth DE (1988) The $\mathrm{T_EX}$book. Addison Wesley, Reading, MA
33. Kotonya G, Sommerville I (1998) Requirements engineering. Wiley, West Sussex, UK
34. Leite JCSP, Franco APM (1993) A strategy for conceptual model acquisition. In: Proc IEEE international symposium on requirements engineering, San Diego, January 1993, IEEE Computer Society, Los Alamitos, CA, pp 243–246
35. Leveson NG (1998) Intent specification: an approach to building human-centered specification. In: Proc third IEEE international conference on requirements engineering, Colorado Springs, CO, IEEE Computer Society, Los Alamitos, CA
36. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: Finkelstein A (ed) The future of software engineering. ACM, Limerick, Ireland
37. Nuseibeh BA (2001) Weaving the software development process between requirements and architecture. In: Proc ICSE2001 Workshop: From software requirements to architectures (STRAW-01), Toronto, May 2001
38. Nuseibeh BA (2001) Weaving together requirements and architecture. IEEE Comp 34:115–117
39. Ossana, J.F (1976) NROFF/TROFF user's manual. Technical report, Bell Laboratories, Murray Hill, NJ, 11 October 1976
40. Ou L (2002) WD-pic, a new paradigm of picture drawing programs and its development as a case study of the use of its user's manual as its specification. Thesis, School of Computer Science, University of Waterloo, Canada
41. Parnas DL, Clements PC (1986) A rational design process: how and why to fake it. IEEE Trans Software Eng SE-12:196–257
42. IEEE (1988) POSIX, IEEE Standard portable operating system interface for computer environments, Technical Committee on Operating Systems of the IEEE Computer Society, IEEE Standard 1003.1-1988
43. Ravid A (1999) A method for extracting and stating software requirements that a user interface prototype contains. Thesis, Faculty of Computer Science, Technion, Haifa, Israelftp://www.cs.technion.ac.il/pub/misc/dberry/alon.ravid/Thesis.doc
44. Shpilberg F (1997) WD-pic, A WYSIWYG, direct-manipulation pic. Thesis, Faculty of Computer Science, Technion, Haifa, Israel
45. Siddiqi J, Shekaran MC (1996) Requirements engineering: the emerging wisdom. IEEE Software 9:15–19
46. Sommerville I, Sawyer P (1997) Requirements engineering, a good practice guide. Wiley, Chichester, UK
47. Swartout W, Balzer R (1982) The inevitable intertwining of specification and implementation. Comm ACM 25:438–440
48. van Lamsweerde A (2000) Requirements engineering in the year 00: a research perspective. In: Proc 22nd international conference on software engineering, Limerick, Ireland, ACM, New York
49. Weidenhaupt K, Pohl K, Jarke M, Haumer P (1998) Scenarios in system development: current practice. IEEE Software 15:34–45
50. Wolfman T (1989) flo—A language for typesetting flowcharts. Thesis, Faculty of Computer Science, Technion, Haifa, Israel
51. Wolfman T, Berry DM (1990) flo—A language for typesetting flowcharts. In: Furuta R (ed) Electronic publishing '90. Cambridge University Press, Cambridge, pp 93–108