What, Not How?

When Is "How" Really "What"? and Some Thoughts on Quality Requirements

Daniel M. Berry
Computer Science Department
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada
dberry@csg.uwaterloo.ca

Abstract

This paper explores the validity of the advice to specify what and not how *for requirements, including quality requirements. In several specific domains, it may be necessary to say How to make the What precise enough.*

Keywords: what, how, requirements specifications, quality requirements, text formatting, robustness, security, safety, survivability, feature conflicts, bagels

Introduction

We have all read requirements that appear to be telling us how to implement a computer based system (CBS) rather than explaining what the CBS is. We have all criticized such requirements documents for digressing to the How of the system and not sticking to the What of the system. Indeed, in the requirements engineering class that I teach at the University of Waterloo, a class that has been taught also by Jo Atlee and Mike Godfrey, the mantra is "What, not how!"

While the mantra seems clear enough, in practice it may be very difficult to separate the Hows from the Whats. Indeed, for some requirements, it may be impossible to specify What without saying something about How. Also there are requirements, usually called *quality* requirements, for which the What specification is simply not very useful, e.g., "The output shall look good.", "The user interface shall be be easy to use.", or "The response time shall be fast." In some of these cases, the only way to make the requirement precise enough to be tested is to say something about how it will be met.

As Michael Jackson observes, putting structure to a problem moves the problem's description at least part of the way towards a solution [13]. I am not minimizing the importance of determining "what *before* how" [13], but I am saying simply that sometimes determining What requires determining How.

Following a brief survey of past discussions of this issue, this paper discusses several specific cases in which How specifications have proved to be essential for a useful, understandable, and testable specification of a requirement, in some cases, a quality requirement:

- good looking formatted text,
- robust, safe, secure, or survivable CBSs,
- non-conflicting features, and
- chewy bagels.

Past Discussions

At a panel on The Role of Software Architecture in Requirements Engineering at the First International Conference on Requirements Engineering in 1994 [26], Michael Jackson spoke for many people when he admitted to dealing with design and implementation issues when he was supposed to be specifying requirements and feeling guilty about the same. However, Jackson did point out that it is necessary to explore feasibility before writing the specifications in order to avoid wasting time on creating specifications that cannot economically be implemented [12]. Nancy Mead argued that software architecture must be considered during requirements engineering to insure that requirements are valid, consistent, complete, and feasible [23]. Colin Potts reminds us that some requirements are infeasible or too expensive to implement. He observes that we can assess feasibility only if we can

relate the requirements to architectural difficulty and that requirements can be ranked by priority only if we understand the architecture of their implementation [24]. Finally, Howard Reubenstein distinguishes between *ideal* and *as-built* architectures and suggests that the former is both necessary and valid to consider during requirements specification [25].

Good Looking Formatted Text

In the Fall of 2000, Donald E. Knuth visited the University of Waterloo campus to be awarded an honorary doctorate and to give a series of lectures. One of these lectures was in fact a town meeting, during which he fielded questions from the audience on the general topic of the art of computer programming. Knuth's answer to my question shed some light on the What vs. How dichotomy in requirements specifications.

A little background is in order. Both Knuth and I have worked in electronic publishing. He is well known for having developed TEX and METAFONT [16, 20, 17, 18]. I have participated in the development of additions to the ditroff [14] family for indexing and for formatting multilingual and multidirectional text [1, 8, 27, 6, 3]. Knuth also participated in the development of a bidirectional version of TEX [19]. Therefore, I was curious as to what he thought was the hardest part of his development of TEX.

Everyone knows Freddy Brooks's famous quotation concerning the difficulty of establishing the requirements of a CBS. "The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements.... No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later." [7]

With this quotation in mind, I asked Knuth, "What was the hardest part of developing TEX?" I was expecting to hear something about deciding what commands TEX shall provide or deciding what TEX shall do for each of these commands. Knuth surprised me by answering that the hardest part was figuring out the line-breaking algorithm.

In text formatters, line breaking is the process of deciding where its text will be broken into lines and deciding how to distribute extra white space between the words of the lines so that each line is both left and right justified. In the following, it is understood that the last so-called word of a line can be only the initial syllables of a word with a hyphen appended. The algorithm used by most formatters prior to TEX can be called *line-by-line greedy*. That is, the line-breaking algorithm of most previous formatters works in a beginning-to-end scan of the text; it fills a line with as many words as possible; then it distributes to the white spaces between the words additional white space equal to the space left over after filling in the words divided evenly by the number of inter-word white spaces. In other words, the algorithm fills each line as much as possible with words and then spreads them out evenly to left and right justify the line. The problem with this traditional line-by-line greedy algorithm is that even though the interword spacing on a line is uniform, the interword spacing of one line can differ wildly from that of another.* When the lines' interword spacings differ wildly and thus visibly, the appearance of the document suffers. Some believe that such poor looking documents are harder to read, since the spacing between words changes from line to line, and the reader has to readjust his or her scan at each line.

Knuth's and Michael Plass's [15] line-breaking algorithm considers each paragraph as a unit and finds the best line breaks in the paragraph, line breaks which when combined with spreading the words of each line out as evenly as possible, yields the most uniform interword spacing over the whole paragraph. With this algorithm, the interword spacing of one line does not differ as wildly from that of its neighbors as it can with the line-by-line greedy algorithm. Basically the algorithm uses dynamic programming to globally maximize the uniformity of or minimize the differences between interword spacings over an entire paragraph. The algorithm scans a paragraph word by word and builds a graph of all possible line breaks in the paragraph. Dynamic programming with this graph allows choosing the line breaks that yield the most uniform interword spacing. In terms of the line-by-line greedy algorithm, the Knuth-Plass algorithm sometimes moves to the following line a word that would otherwise fit into the current line. This move causes the uniform interword gaps on the two lines to be closer to equal than it would be if the word were put in the current line. Putting the word in the current line would cause the spacing of the next line to be much larger than the spacing on the current line.

How successful is the Knuth-Plass algorithm? A glance at any TEX-formatted document shows how much more uniform the interword spacing is than in any document formatted by ditroff, which uses a line-by-line greedy algorithm. The superiority of the Knuth-Plass algorithm is most apparent when comparing a document formatted in TEX to one formatted in Microsoft's Word, which cannot get the interword spacing uniform even

* You can see this effect in the document you are reading. Wildly varying interword spacing is unavoidable when the column of text is narrow relative to the point size of the text.

within one line!

When I heard Knuth's answer, I thought to myself, "Oh well, the perennial, quintessential super programmer turned mathematician and algorithm theoretician (or maybe it is the other way around!) is always concerned with algorithms, with the How of software. Probably, he considered the requirements of TEX to be obvious and straightforward." I kept my opinion to myself.

Later, I began to think carefully about Knuth's answer. I eventually realized that what I thought was a How issue was really a What issue, that Knuth and Plass had used a How specification to specify a What that would otherwise not be specifiable at all. In the following, I am presenting *my* theory as to what Knuth's thoughts were at the time he was designing TEX's line-breaking algorithm.* As Knuth was considering the requirements for TEX, that is, *what* it shall do, he decided that he wanted left and right justification as the norm and the interword spacing to look better than it does in all extant algorithmic formatters. He had seen the output of earlier formatters such as ditroff and compared it to traditional hand-typeset books and had observed that the hand-typeset material looked far better.

That the output looks good is considered a quality requirement. However, Knuth was not content to leave looking good as unspecified and to hope that an implementor will get inspired and write an implementation that produces output that looks good. He wanted to define precisely what it means to look good. He knew that the line-by-line greedy algorithm is easy to specify, both algorithmically, as before, and declaratively: On the basis of a beginning-to-end scan of the text, each line contains as many words that can fit, and the words in each line are spread apart evenly to achieve left and right justification. However, looking good is specifiable neither algorithmically nor declaratively unless it is known what it means.

Knuth explored what human typesetters do to produce the traditional high-quality typeset material that looks good. He learned that they do not always follow the greedy algorithm. Sometimes, a typesetter can see a little ahead or learns by trial and error that if he or she fills the current line as much as possible, then the interword spacing on the next line will be observably bigger. He or she then moves the last word of the current line to the next line, causing the line spacing on the current line to increase, and after filling and spreading the new next line, its spacing is about the same as that of the current line. A good typesetter with a good eye for spacing can sometimes see over a whole paragraph line breaks that yield good uniform spacing over the whole paragraph. However, at this point, line breaking has become an art.

The fact that human typesetters do achieve good looking output means that it is possible to create good looking output. The fact that it is an art does not bode well for an algorithm to consistently produce good output. With some inspiration, Knuth and Plass began to see the similarity of the line breaking algorithm with other global optimization problems. There is a large, but finite number of ways to break a paragraph of words into lines. One considers each interword gap and hyphenation point as a possible line break point. The problem then is to find the choice of line break points such that if the text on each line is spread out evenly, the interword spacing on all line are the closest to being equal. Stated this way, it is clear that line breaking is one example of a class optimization problems that require exponential time to solve algorithmically. The most famous of these is the traveling salesperson problem, to find the shortest route covering all places the salesperson must visit. After this realization, it was relatively straightforward to arrive at the dynamic programming algorithm described in the Knuth and Plass paper [15].

It is instructive to notice why the algorithm is called a paragraphing algorithm. Clearly, one could do the optimization of the interword spacing over an entire document. However, then the computational costs would be prohibitive. The problem is to identify a unit of text small enough that the algorithm, requiring exponential time and space in the general case, can be brought down to requiring reasonable time and space, but enough larger than a line that uniformity of spacing over the lines of the unit is observable. The paragraph, normally consisting of several lines, is a good unit over which to apply the algorithm. Hence, the line breaking algorithm of Knuth and Plass is called a paragraphing algorithm.

In the end, TEX has output that looks good, that looks better than that produced by other formatters, such as ditroff, only because Knuth has found a way to implement it. In effect, the discovery of the algorithm served as a validation of the feasibility of requiring more uniform spacing than is possible with line-by-line greediness and that is like what is achievable by skilled human typesetters. In fact, since the algorithm finds an optimal choice of line breaks, the uniformity of the spacing in paragraphs is the best it can be and may be even better than is produced by many human typesetters.

The way to do line-breaking then becomes the specification of the requirements; that is, the How becomes the

*As of this writing, I have not confirmed this theory with Knuth. I will do so for a later version of this paper. If you read a later version in which this footnote is gone, then Knuth has confirmed my theory.

What. Consequently, all* implementations of TEX are required to produce the same uniformly spaced output that the algorithm produces. However, even this How is not really a How. Implicitly, any algorithm that yields the same output is acceptable, particularly if it performs better than the current output.§

Still, it appears that Knuth has gone far beyond what is considered acceptable in exposing implementation details to the user, in deciding that a particular algorithm shall be the standard one used by any implementation of TEX. Knuth has gone so far as to describe the algorithm and its implications in the TEX user's manual, *The TEXbook*. This user's manual serves as the requirements specification [7] for TEX. However, look at the alternatives.

One alternative is to leave the quality requirement specified only in words. In this case, the requirement is one of:

1. The output looks good.
2. The interword spacing looks good.
3. The interword spacing looks better than in any other existing formatter.
4. The interword spacing looks as good as produced by the best human typesetters.

It is clear that neither of these says much about what actually happens and all of them leave too much leeway to the implementor.

Another alternative is to specify declaratively the effect of the algorithm:

1. The interword spacing is as uniform as possible over each paragraph.
2. The interword spacing is that achieved by minimizing the differences between the interword spacing over each paragraph, assuming that the spacing on each line is completely uniform.

The problem with such declarative specifications is that there is no guarantee that it is feasible or even possible to implement the requirement. It is possible to specify requirements that are impossible to implement, e.g., "The program accepts a program *p* and a file *f* as input and decides whether *p*, presented with input *f*, halts." It is also possible to specify requirements that are simply not feasible given the available technology, e.g., "The program produces interword spacing that is as uniform as possible over the entire input document of any size." Giving a specific algorithm for which one has done a complexity analysis and provided multiplicative constants allows determination of possibility and feasibility. Even such an analysis may not be enough. Nancy Day observes that is not uncommon to write infeasible requirements for large and real-time CBSs because we do not have enough experience building CBSs in the domain at hand.

As a side effect, Knuth ended up precisely defining the normally unspecifiable quality attribute of "looking good" in a way that captured the essence of looking good in the domain of line-breaking in typesetting. In this case, the How specification ends up being a highly sought-after What specification.

In the end, it was clear that Knuth's answer to my question *was* about requirements, and that what he thought was the hardest part of building TEX was deciding requirements.

Robust, Safe, Secure, or Survivable CBSs

Other requirements for which How specifications are necessary and are accepted as What specifications are system robustness, system safety, system security, and system survivability. System security is taken as the representative of these domains, because the way to deal with these sorts of requirements is the same. System security was the first of these requirements to be considered formally, back in the 1970s and 1980s [9].

It was learned fairly early in the game that it was impossible to add security at implementation time or after deployment to an insecure system or one specified and designed totally oblivious to security requirements. Security had to be required in from the beginning, and consideration of security requirements had to permeate the entire lifecycle from requirements analysis, through design and implementation, to testing. Otherwise, it was too easy to build a system whose very design precludes security. An example of such a system is the Internet, whose very open design precludes the total security that e-commerce demands. Security concerns must also be considered carefully during maintenance, lest a modification inadvertently negate all previously provided security mechanisms.

It is fairly easy to give an over-all requirement for a

---

*Admittedly, Knuth has arranged that there is, in fact, only *one* implementation of TEX at any time, the current standard, produced only by Knuth. He has gone further and decreed that once he is no longer able to work on TEX, the last version he has produced is the last version ever, and that all remaining bugs become features.

§ However, as Michael Jackson points out, there is a hidden, spurious circularity here. The author of such a specification is saying, "I can't explain the specification behaviour in purely external terms, so I need to postulate some internal state functions. But don't you worry about the internal states I have described: just ignore them and pay attention to the behaviour they induce (which is, of course, exactly what I am admitting I can not do myself).

secure CBS, that it release no data to any user except one authorized by the current security policy to receive them. However, without showing how specific parts of the CBS do their activities, it is impossible to validate that the CBS as specified will indeed satisfy its security requirements after deployment. Unfortunately, it is often impossible to see a security leak until enough detail is understood about implementation to see how the data are moved around and where the potential leaks are, sometimes, through covert channels. This impossibility creates a dilemma. The only way to ensure security of a CBS is to require it from the beginning. Since requirements specifications are not supposed to describe how to implement the CBS, the specification remains at the What level. However, it is impossible to see where the security leaks are without considering implementation details. Thus, in the absence of a How specification, it cannot be verified that the specified system is indeed as secure as required. It might even be verifiable that the What specification does meet the security requirements simply because the details by which the specified system becomes insecure have not been exposed in the specification. Only a How specification expresses these details. Thus, in the security domain, the How of the CBS becomes essential for a What specification.

The same dilemma [5] exists for specifiers of robust, safety-critical, and survivable systems. Reliability, safety, and survivability must be required from the beginning, but the potential errors and hazards cannot be known fully until implementation details are given [21, 10, 2, 11, 22].

Non-Conflicting Features

In telephony and other reactive CBSs to which features are constantly being added, it is necessary to investigate any proposed new feature to see

- if it interacts any existing feature and
- to make sure that the new feature does not conflict with any existing feature.

We want to be able to do the analysis based on specifications that are as high level as possible, preferably purely What specifications that state only *what* each feature is. However, interaction and conflict occur because at least two features *f1* and *f2* use the same hardware or the same data *D* to achieve their function. The use of *D* by *f1* in some way affects or is affected by the use of *D* by *f2*.

In the end, all such interactions and conflicts can be detected if the detailed code of all features is examined to see to which devices or data the features share access. However, no one wants to have to dig so deeply into the code. To be able to detect interaction and conflict from high-level specifications, it must be apparent from the specifications that *f1* and *f2* share access to some *D*.

If the specification is too high level, it may be that the specifications of *f1* and *f2* share no words in common. Therefore, it might appear that they do not interact when they do. For example,

- *unlisted phone number*, which allows a subscriber not to list his or her phone number, and
- *caller identification*, which allows a callee to see the name and number of a caller,

conflict because caller identification shows a name with the phone number, contrary to what the subscriber with an unlisted phone number wants. With a high-level What specification such as given in the phrases beginning with "which" in the bulleted list, the conflict might not be apparent because the given specification of *unlisted phone number* neglects to mention that the goal is to avoid listing the phone number with the subscriber's name.

Alternatively, it may be that the specifications of *f1* and *f2* share words in common, but those words do not represent any shared *D*. In the following examples, consider a feature's name to be the highest-level What specification possible. Consider telephony features with the word "call" in their names:

- *Call Forwarding On Busy*
- *Call Forwarding No Answer*
- *Call Waiting*
- *Originating Call Screening*
- *Three-Way Calling*

In the four requirements that use "call" as a noun, the requirement is a verb phrase that says that something is happening to the call. In the other case, "call" is the root of a gerund describing how the call is made. So, in a sense, all five share data, the data being the call itself. *Call* is a high-level datum containing all information involved in making a call. So in this sense, they all interact and possibly conflict. However, declaring all features to conflict is overkill. We need to go to more detailed levels to answer the question more precisely.

In fact, each of the pairs:

- *Call Waiting* and *Call Forwarding No Answer*
- *Originating Call Screening* and *Call Waiting*

does not conflict, and each of the pairs:

- *Call Waiting* and *Call Forwarding on Busy*
- *Call Waiting* and *Three-Way Calling*

does conflict.

It is impossible to detect these conflicts and non-conflicts without going into greater detail about the functionality, sometimes, as far as having to work with a How specification. The key is to describe only enough of the How to detect the conflicts, in order to be able to specify

what is and what is not.

A thorough analysis of a proposed *call forwarding* feature requires a full analysis of the problem [13] in the real world of people, vacations, office, moving from office to office, delegating responsibility, and when to forward, in order to learn the distinction between *follow-me* and *delegation* forwarding and between forwarding *always*, *on no answer*, and *on busy* [28]. However, then, completing the analysis requires investigating enough of the implementation in order to identify interaction and conflicts with features already available. Here again, What specifications are inadequate for answering requirements-related questions. How specifications are needed in order to answer these questions.

Chewy Bagels

Sometimes the best or only way to specify a desired effect is to describe one way to achieve it, especially if there is only one way to achieve it. Good looking formatted text is one such case. A New York bagel is another. How many of you readers have ever really had one? A New York bagel is not just a baked good with a hole in it, despite the widespread proliferation of places that make just that and call it a bagel in order to profit from the current bagelmania. A donut is another baked good with a hole it in, and we all know that a bagel and donut have nothing in common except the hole; indeed, a bagel and a donut have literally nothing in common!

I have a T-shirt from one producer of New York bagels, Bruegger's Bagels, that gives a specification of bagelhood. The T-shirt has imprinted on it a blueprint, captioned "Building a Better Bagel", showing the physical dimensions of the perfect New York bagel, from Bruegger's of course. The blueprint, shown in Figure 1,* gives for the bagel a plan, an elevation, and a cross section along an axis perpendicular to a vertical line drawn in the plan. However, a donut satisfies these physical dimensions. To distinguish a New York bagel from any other baked good with a hole, Detail 1-A of the blueprint, shown in Figure 2, has specifications of the elasticity of the surface and the moisture content of the inside. The inner mantle should yield without breaking up to 45 pounds per square inch stress, and the inside should maintain a 20 to 25 percent moisture content. These surface elasticity and inner moisture content specifications together specify the chewiness of a New York bagel. A donut does not satisfy this chewiness specification. In place of chewiness specification, one could use the famous teething definition, that a proper bagel can be used by a baby for teething for at least 10 minutes without disintegrating into a ball of mush [4]. A donut clearly fails this specification.

Is this chewiness an essential requirement of a New York bagel? I think so, because without the chewiness, the baked good with a hole in it is not a New York bagel. It is a bread with a hole, a donut, or perhaps, another kind of bagel entirely, such as Montréal or Northern.

As nice as these What specifications are, the simplest description of a New York bagel is algorithmic:

1. Use high gluten flour dough that has risen.
2. Make a ring with outer diameter 4 inches and inner diameter 1 inch and with a cross section of 1.5 inches in diameter.
3. Put the ring into boiling water for 30 seconds.
4. Bake the ring at 400°F for 10 minutes or until golden brown.

The step that is left out or changed by most poor imitations of New York bagels and other kinds of bagels is Step 3.

A What specification is normally preferred to a How specification because the What specification says only what is desired and allows the implementor the freedom to achieve the requirements in any way he or she can. It spurs competition to find more efficient ways to achieve what is specified than originally conceived. Some of the companies that fail to make genuine New York bagels do so because they have decided to make different kind of bagels. These companies include the makers of Montréal bagels, a different kind of bagel with its own fans. Others that fail to make genuine New York bagels do so because they have decided that the boiling is unnecessary; they do not get the proper surface yield or interior moisture content. They make bread with a hole. Many supermarket-made bagels are in this category. Finally, others, e.g., the Great Canadian Bagel, McDonald's and Tim Horton's (if the words of clerks working there can be believed) have tried steaming in place of boiling. It almost works, but the surface yield is not high enough and the interior moisture content is too high. Note the headline that appeared in the San Jose *Mercury News*, Sunday, April 25, 1999, after McDonald's introduced its steamed bagels: "McDonald's bagels are steamed; so are purists" Among the steamed purists was none other than Eli Zabar of New York's famous Zabar's Delicatessen, one of the prime sources of genuine New York bagels along with H&H!

These attempts to steam are clearly examples of trying to find a cheaper way to achieve the What specification than the standard How method. However, to date no algorithm other than the standard How method has succeeded to achieve the desired What specification. Here is an

---

* The blueprint is copyright by and is a registered trademark of Bruegger's Bagels Corporation and is reprinted with permission.

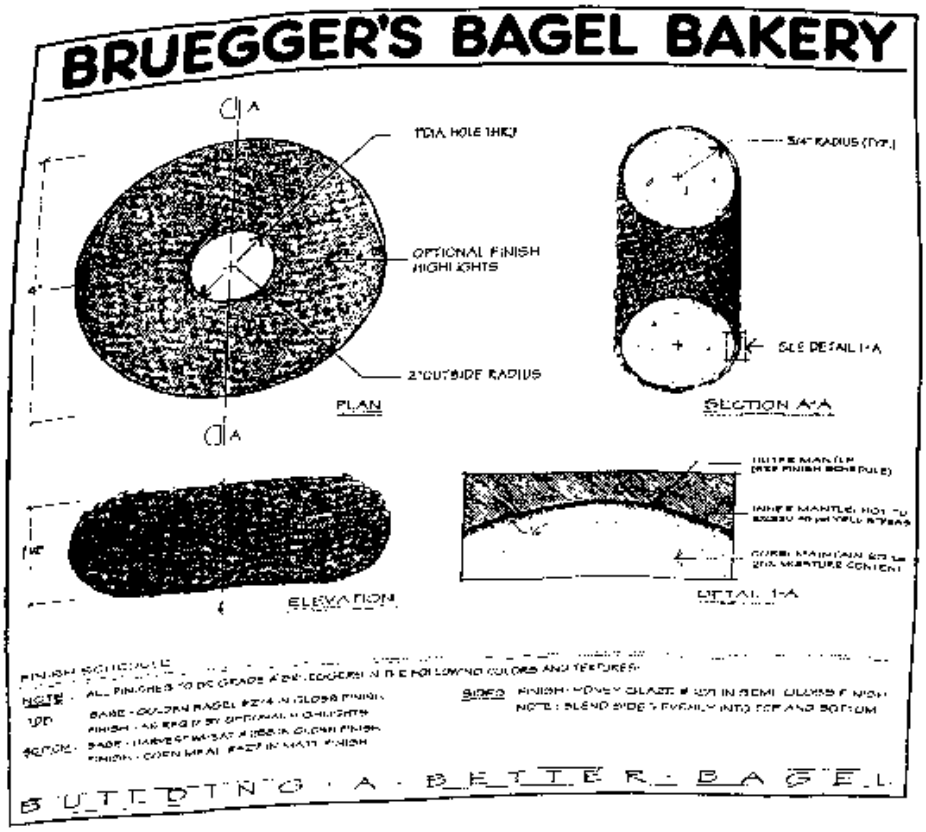

Figure 1: Blueprint of Bruegger's Bagel

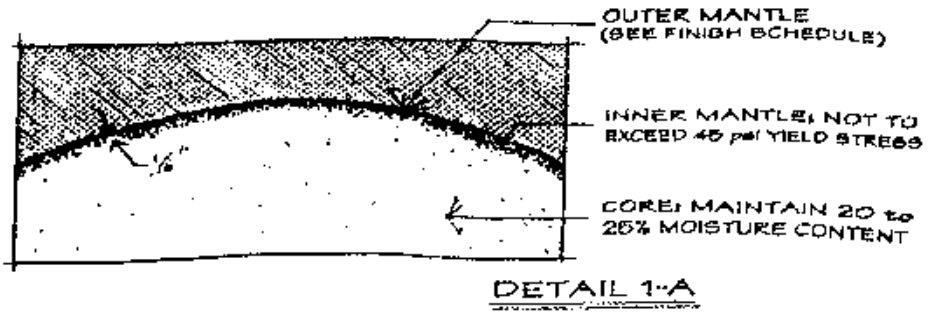

Figure 2: Detail 1-A from the Blueprint of Bruegger's Bagel

example in which a How method specification may be better than a What specification. It is certainly simpler in the sense that it is easier to tell what needs to be done. However, in opting for this How specification, one is discouraging innovation. However, in the case of the New York bagel, perhaps innovation *should* be discouraged!

An algorithmic description is the clearest, simplest specification of what a New York bagel is. While it does prescribe how to make it, in principle anything that tastes and feels the same will be accepted as a bagel. Unfortunately for those who wish to optimize on the time to produce a bagel and to eliminate the need for a boiling vat, all other ways tried so far have yet to produce exactly the desired taste and feel.

Conclusion

This paper has considered several situations in which design and implementation details are necessary to resolve issues that should be resolved during specification. There are others for sure, for example, in hardware architectures, but space does not permit their exploration. The conclusion after consideration of these examples is that sometimes, a How specification is significantly clearer or briefer than a What specification and that sometimes, a How specification is needed for information that is not available in a What specification. Also, sometimes, a How specification is needed to make q quality requirement precise enough to be tested. Therefore, when it is appropriate to use a How specification, do so without guilt and in good health!

Acknowledgments

I thank Martin Feather for pointing out that my T-shirt had a What specification of bagels and that sometimes a How specification is better. I thank Michael Jackson for an interesting e-mail discussion on What vs. How. Finally, I thank Jo Atlee, Nancy Day, Mike Godfrey, and Leah Goldin for valuable comments on earlier drafts of this paper.

References

[1] Abe, K.K. and Berry, D.M., "indx and findphrases, A System for Generating Indexes for Ditroff Documents," *Software—Practice and Experience* **19**(1), p.1–34, 1989.

[2] Anderson, T., de Lemos, R., and Saeed, A., "Analysis of Safety Requirements for Process Control Systems," in *Predictably Dependable Computing Systems*, ed. B. Randell, J.C. Laprie, B. Littlewood, H. Kopetz, Springer, Berlin, 1995.

[3] Becker, Z. and Berry, D.M., "triroff, an Adaptation of the Device-Independent troff for Formatting Tri-Directional Text," *Electronic Publishing* **2**(3), p.119–142, October 1990.

[4] Berman, C. and Munshower, S., *Bagelmania: the Hole Story*, HP Books, Tucson, AX, 1987.

[5] Berry, D.M., "The Safety Requirements Engineering Dilemma," in *Proceedings of the Ninth International Workshop on Software Specification and Design (IWSSD'98)*, Ise Shima, Japan, 16-18 April 1998.

[6] Berry, D.M., "Stretching letter and slanted-baseline formatting for Arabic, Hebrew, and Persian with ditroff/ffortid and Dynamic POSTSCRIPT Fonts," *Electronic Publishing, Origination, Dissemination, and Design*, 1998, to appear.

[7] Brooks, F.P. Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley, Reading, MA, 1995, Second Edition.

[8] Buchman, C., Berry, D.M., and Gonczarowski, J., "*DITROFF/FFORTID*, An Adaptation of the UNIX *DITROFF* for Formatting Bi-Directional Text," *ACM Transactions on Office Information Systems* **3**(4), p.380–397, October 1985.

[9] Cheheyl, M.H., Gasser, M., Huff, G.A., and Millen, J.K., "Verifying Security," *Computing Surveys* **13**(3), p.279–340, September 1981.

[10] de Lemos, R., Saeed, A., and Anderson, T., "On the Safety Analysis of Requirements Specifications," p. 217–227 in *Proceedings of the Thirteenth International Conference on Safety, Reliability, and Security (SAFECOMP '94)*, ed. V. Maggioli, Springer, 1994.

[11] de Lemos, R., Saeed, A., and Anderson, T., "On the Integration of Requirements Analysis and Safety Analysis for Safety-Critical Software," Technical Report, Department of Computer Science, University of Newcastle upon Tyne, UK, 1998.

[12] Jackson, M.A., "The Role of Architecture in Requirements Engineering," p. 241 in *Proceedings of the First International Conference on Requirements Engineering*, IEEE Computer Society, Colorado Springs, CO, April 18–22 1994.

[13] Jackson, M.A., *Problem Frames: Analysing and Structuring Software Development Problems*, Addison-Wesley, Harlow, England, 2001.

[14] Kernighan, B.W., "A Typesetter-independent TROFF," Computing Science Technical Report No. 97, Bell Laboratories, Murray Hill, NJ, March 1982.

[15] Knuth, D.E. and Plass, M.F., "Breaking Paragraphs into Lines," *Software—Practice and Experience* **11**, p.1119–1184, 1981.

[16] Knuth, D.E., *Computers & Typesetting, Volume B: TEX: The Program*, Addison Wesley, Reading, MA, 1986.

[17] Knuth, D.E. and Bibby, D., *Computers & Typesetting, Volume D: Metafont: The Program*, Addison Wesley, Reading, MA, 1986.

[18] Knuth, D.E., *The Metafont Book*, Addison Wesley, Reading, MA, 1986.

[19] Knuth, D.E. and MacKay, P., "Mixing Right-to-left Texts with Left-to-right Texts," *TUGboat* **8**(1), p.14–25, 1987.

[20] Knuth, D.E., *The TEXbook*, Addison Wesley, Reading, MA, 1988.

[21] Leveson, N.G., *Safeware: System Safety and Computers*, Addison Wesley, Reading, MA, 1995.

[22] Linger, R.C., Mead, N.R., and Lipson, H.F., "Requirements Definition for Survivable Network Systems," p. 14–23 in *Proceedings of the Third International Conference on Requirements Engineering*, IEEE Computer Society, Colorado Springs, CO, 1998.

[23] Mead, N.R., "The Role of Software Architecture in Requirements Engineering," p. 242 in *Proceedings of the First International Conference on Requirements Engineering*, IEEE Computer Society, Colorado Springs, CO, April 18–22 1994.

[24] Potts, C., "Three Architectures in Search of Requirements," p. 243 in *Proceedings of the First International Conference on Requirements Engineering*, IEEE Computer Society, Colorado Springs, CO, April 18–22 1994.

[25] Reubenstein, H.B., "The Role of Software Architecture in Requirements Engineering," p. 244 in *Proceedings of the First International Conference on Requirements Engineering*, IEEE Computer Society, Colorado Springs, CO, April 18–22 1994.

[26] Shekaran, M.C., "Panel: The Role of Software Architecture in Requirements Engineering," p. 239–245 in *Proceedings of the First International Conference on Requirements Engineering*, IEEE Computer Society, Colorado Springs, CO, April 18–22 1994.

[27] Srouji, J. and Berry, D.M., "Arabic Formatting with ditroff/ffortid," *Electronic Publishing* **5**(4), p.163–208, December 1992.

[28] Zave, P., "Secrets of Call Forwarding: A Specification Case Study," p. 153–168 in *Formal Description Techniques VIII, Proceedings of the Eighth International IFIP Conference on Formal Description Techniques for Distributed Systems and Communications Protocols*, Chapman & Hall, 1996.