Precise Documentation: The Key To Better Software

David Lorge Parnas

Middle Road Software, Inc.

Abstract. The prime cause of the sorry "state of the art" in software development is our failure to produce good design documentation. Poor documentation is the cause of many errors and reduces efficiency in every phase of a software product's development and use. Most software developers believe that "documentation" refers to a collection of wordy, unstructured, introductory descriptions, thousands of pages that nobody wanted to write and nobody trusts. In contrast, Engineers in more traditional disciplines think of precise blueprints, circuit diagrams, and mathematical specifications of component properties. Software developers do not know how to produce precise documents for software. Software developments also think that documentation is something written after the software has been developed. In other fields of Engineering much of the documentation is written before and during the development. It represents forethought not afterthought. Among the benefits of better documentation would be: easier reuse of old designs, better communication about requirements, more useful design reviews, easier integration of separately written modules, more effective code inspection, more effective testing, and more efficient corrections and improvements. This paper explains how to produce and use precise software documentation and illustrate the methods with several examples.

1 Documentation: a perpetually unpopular topic

This paper presents the results of many years of research on a very unpopular subject. Software documentation is disliked by almost everyone.

- Program developers don't want to prepare documentation; their lack of interest is evident in the documents that they deliver.
- User documentation is often left to technical writers who do not necessarily know all the details. The documents are often initially incorrect, inconsistent and incomplete and must be revised when users complain.
- The intended readers find the documentation to be poorly organized, poorly prepared and unreliable; they do not want to use it. Most consider "try it and see" or "look at the code" preferable to relying on documentation.
- User documentation is rapidly being replaced by "help" systems because it is hard to find the details that are sought in conventional documentation. Unfortunately, the "help" system answers a set of frequently occurring questions and is usually incomplete and redundant. Those who have an unusual question don't get much help.

These factors feed each other in a vicious cycle.

- Reduced quality leads to reduced usage.
- Reduced usage leads to reductions in both resources and motivation.
- Reduced resources and motivation degrade quality further.

Most Computer Science researchers do not see software documentation as a topic within computer science. They can see no mathematics, no algorithms, and no models of computation; consequently, they show no interest in it. Even those whose research approach is derived from mathematics, do not view what they produce as documentation. As a consequence of failing to think about how their work could be used, they often produce notations that are not useful to practitioners and, consequently, not used.

This paper argues that the preparation of well-organized, precise documentation for use by program developers and maintainers is essential if we want to develop a disciplined software profession producing trustworthy software. It shows how basic mathematics and computer science can be used to produce documentation that is precise, accurate, and (most important) useful. The first section of the paper discusses why documentation is important. The main sections show how we can produce more useful documents. The final section discusses future research and the transfer of this research to practice.

2 Types of documents

Any discussion of documentation must recognize that there are many types of documents, each useful in different situations. This section introduces a number of distinctions that are important for understanding the basic issues.

2.1 Programming vs. software design

Newcomers to the computer field often believe that "software design" is just another name for programming. In fact, programming is only a small part of software development [30]. We use the term "software" to refer to a program or set of programs written by one group of people for repeated use by another group of people. This is very different from producing a program for one's own use or for a single use.

- When producing a program for one's own use, you can expect the user to understand the program and know how to use it. For example, that user (yourself) will be able to understand the source of an error message or the cause of some other failure. There is no need to offer a manual that explains what parameters mean, the format of the input, or compatibility issues. All of these are required when preparing a program that will be used by strangers. If the user is the creator, questions that arise because he or she has forgotten the details can be answered by studying the code. If a non-developer user forgets something that is not in the documents, they may have no alternative but to find someone who knows the code.

– When producing a program for a single use, there is no need to design a program that will be easy to change or easily maintained in several versions. In contrast, programs that are successful software products will be used for many years; several versions may be in use simultaneously. The cost of updating and improving the program often greatly exceeds the original development costs. Consequently, it is very important to design the program so that it is as easy to change as possible and so that members of the software product line have as much in common as possible. These commonalities must be documented so that they remain common as individual family members are revised.

It is the differences between software development and programming (multi-person involvement, multi-version use) that make documentation important for software development.

2.2 What is a document?

A *description* of a system is accurate information about the system in written or spoken form. Usually, we are concerned with written descriptions, descriptions that will exist for a substantial period of time.

A *document* is a written description that has an official status or authority and may be used as evidence. In development, a document is usually considered binding, i.e. it restricts what may be created. If deviation is needed, revisions of the document must be approved by the responsible authority.

A document is expected to be correct or accurate, i.e. it is expected that the information that one may glean from the document is actually true of the system being described. When this is not the case, there is an error: either the system is defective, the document is incorrect, or both are wrong. In any case, the purported document is not a description of the system. Such faulty descriptions are often still referred to as "documents" if they have an official status.

Documents are expected to be precise and unambiguous. In other words, there should be no doubt about what they mean. However, even if they are imprecise or unclear, they may still be considered to be documents.

It should be noted that individual documents are not expected to be complete in the sense of providing all of the information that might be given about the system. Some issues may not have been resolved at the time that a document was written or may be the subject of other documents[1]. However, the content expected in a class of documents should be specified. For example, food packages are required to state certain specific properties of their contents. When there is a content-specification, documents are expected to be *complete relative to that specification*.

In connection with software, the word "document" is frequently used very sloppily. Information accompanying software often comes with disclaimers that warn that they are not necessarily accurate or precise descriptions of the software. Where one might expect statements of responsibility, we find disclaimers.

[1] It is generally bad practice to have the same information in more than one document as this allows the documents to become inconsistent if some are changed and others are not.

2.3 Are computer programs self-documenting?

One of the reasons that many do not take the problem of software documentation seriously is that the code itself looks like a document. We view it on paper or on a screen using exactly the same skills and tools that we would use for a reading a document. In 2006, Brad Smith, Microsoft Senior Vice President and General Counsel, said. "The Windows source code is the ultimate documentation of Windows Server technologies" [33].

No such confusion between the product and the documentation occurs when we develop physical artifacts; there is a clear difference between a circuit diagram and the circuit or between a bridge and its documentation. We need separate documents for physical products such as buildings and electronic circuits. Nobody wants to crawl around a building to find the locations of the ducts and pipes. We expect to find this information in documents.[2]

It is quite common to hear code described as self documenting; this may be true "in theory" but, in practice, most programs are so complex that regarding them as their own documentation is either a naive illusion or disingenuous. We need documents that extract the essential information, abstracting from the huge amounts of information in the code that we do not need. Generally, what is needed is not a single document but a set of documents, each giving a different view of the product. In other words, we have to ignore the fact that we could print out the code as a set of characters on some medium and provide the type of documentation that is routinely provided for physical products.

2.4 Internal documentation vs. separate documents

With physical products it is clear that we do not want the documentation distributed throughout the product. We do not want to climb a bridge to find out what size the nuts and bolts are. Moreover, we do not want to have to understand the bridge structure to decide whether or not we can drive our (heavily loaded) vehicle across. We expect documentation to be separate from the product and to provide the most needed abstractions (e.g. capacity).

Some researchers propose that specifications, in the form of assertions [12] or program functions [11], be placed throughout the code. This may be useful to the developers but it does not meet the needs of other readers. Someone who wants to use a program should not have to look in the code it to find out how to use it or what its capabilities and limits are. This is especially true of testers who asked to prepare "black box" test suites in advance of completion so that they can start their testing as soon as the code is deemed ready to test.

[2] Our expectations are not always met; sometimes someone has not done their job.

2.5 Models vs. documents

Recognition of a need for something other than the code has sparked an interest in models and approaches described as model-driven engineering. It is important to understand the distinction between "model "and "document".

A *model* of a product is a simplified depiction of that product; a model may be physical (often reduced in size and detail) or abstract. A model will have some important properties of the original. However, not all properties of the model are properties of the actual system. For example, a model airplane may be made of wood or plastic; this is not true of the airplane being modelled. A model of a queue may not depict all of the finite limitations of a real queue. In fact, it is common to deal with the fact that we can build software stacks with a variety of capacity limits by using a model that has unlimited capacity. No real stack has this property.

A *mathematical model* of a system is a mathematical description of a model; that mighty not actually exist. A mathematical model may be used to calculate or predict properties of the system that it models.

Models can be very useful to developers but, because they are not necessarily accurate descriptions, they do not serve as documents. One can derive information from some models that will not be true of the real system. For example, using the model of an unlimited stack, one may prove that the sequence PUSH;POP on a stack will leave it unchanged. This theorem is not true for real stacks because they might be full before the PUSH and the subsequent POP will remove something that was there before.

It follows that models must be used with great care; the user must always be aware that any information obtained by analyzing the model might not apply to the actual product. Only if the model has no false implications, can it be treated as a document.

However, every precise and accurate document can serve as a safe mathematical model because a document is usually simpler than the real product and has the property that everything you can derive from the document is true of the actual object. Precise documents can be analyzed and simulated just as other models can.

2.6 Design documents vs. introductory documentation

When a document is prepared, it may be intended for use either as a tutorial narrative or as a reference work.

- Tutorial narratives are usually designed to be read from start to end.
- Reference works are designed to help a reader to retrieve specific facts from the document.
- Narratives are usually intended for use by people with little previous knowledge about the subject.
- Reference documents are generally designed for people who already know a lot about the subject but need to fill specific gaps in their knowledge.

The difference may be illustrated by contrasting introductory language textbooks with dictionaries for the same language. The textbooks begin with the easier and more

fundamental characteristics of the language. Dictionaries arrange words in a specified (but arbitrary) order that allows a user to quickly find the meaning of a specific word.

Generally, narratives make poor reference works and reference works are a poor way to get an introduction to a subject.

In the software field we need both kinds of documents. New users and developers will need to receive a basic understanding of the product. Experienced users and developers will need reference documents.

This paper is about reference documents. The preparation of introductory narratives is a very different task.

2.7 Specifications vs. other descriptions

When preparing engineering documents, it is important to be conscious of the role that the document will play in the development process. There are two basic roles, description and specification.

- *Descriptions* provide properties of a product that exists (or once existed).
- S*pecifications*[3] are descriptions that state only required properties of a product that might not yet exist.
- A specification that states all required properties is called a *full specification*.
- General descriptions may include properties that are incidental and not required.
- Specifications should only state properties that are required. If a product does not satisfy a specification it is not acceptable for the use intended by the specification writer.

The difference between a specification and other descriptions is one of intent, not form. Every specification that a product satisfies is a description of that product. The only way to tell if a description is intended to be interpreted as a specification is what is said about the document, not its contents or notation.

The distinction made here is important whenever one product is used as a component of another one. The builder of the using product should be able to assume that any replacements or improvements of its components will still have the properties stated in a specification. That is not true if the document is a description but not a specification. Unfortunately, many are not careful about this distinction.

A specification imposes obligations on the implementers, users, and anyone who requests a product that meets the specification.

When presented with a specification, *implementers* may either

- accept the task of implementing that specification, or
- reject the job completely, or
- report problems with the specification and propose a revision.

They may not accept the task and then build something that does not satisfy the specification.

[3] In this paper, we use the word “specification” as it is traditionally used in Engineering, which is different from the way that it has come to be used in Computer Science. In Computer Science, the word is used, often without definition, to denote any model with some (often unspecified) relation to a product.

- *Users* must be able to count on the properties stated in a specification; they must not base their work on any properties stated in another description unless they are also stated in the specification.
- *Purchasers* are obligated to accept (and pay for) any product that meets the (full) specification included in a purchase agreement or bid.

Other descriptions may be useful for understanding particular implementations, for example additional descriptive material may document the behaviour of an implementation in situations that were treated as "don't care" cases in the specification.

2.8 Extracted documents

If the designers did not produce the desired documentation, or if the document was not maintained when the product was revised, it is possible to produce description documents by examining the product. For example, the plumbing in a building may be documented by tracing the pipes. If the pipes are not accessible, a die can be introduced in the water and the flow determined by observing the colour of the water at the taps.

There are many things that documentation extracted from the product cannot tell you. Documents produced by examining a product will be descriptions but not usually specifications. Extracted documents cannot tell you what was intended or what is required. Describing the actual structure is valuable in many circumstances but not a substitute for a specification.

Extracted documents usually contain low-level facts rather than the abstractions that would be of more value. The derivation of useful abstractions from construction details is a difficult problem because it requires distinguishing between essential and incidental aspects of the code.

Extracted documentation is obviously of no value during development, e.g. for design reviews or to improve communication between developers. It is also not a valid guide for testers because one would be testing against a description, not a specification. Testing on the basis of derived documents is circular; if you do this, you are assuming that the code is correct and testing to see that it does what it does.

Because software is always in machine-readable form, there are many tools that process code, extracting comments and declarations, and assemble these abstracts into files that are called documents. These extracted documents mix irrelevant internal details with information that is important for users; these documents will not usually be good documents for users, and reviewers, or maintainers. They include mostly the nearly random selection of information that a programmer chose to put in the comments.

These tools are apparently intended for people who are required to produce documentation but do not want to do the work required. Their major advantage is that the documentation produced can be revised quickly and automatically whenever the program is changed.

3 Roles played by documents in development

There are a number of ways that precise documentation can be used during the development and maintenance of a software product. The following sections describe the various ways that the software will be used.

3.1 Design through documentation

Design is a process of decision making; each decision eliminates one or more alternative designs. Writing a precise design document forces designers to make decisions and can help them to make better ones. If the documentation notation is precise, it will help the designers to think about the important details rather than evade making conscious decisions. A design decision will only be able to guide and constrain subsequent decisions if it is clearly and precisely documented. Documentation is the medium that designers use to record and communicate their decisions [6].

3.2 Documentation based design reviews

Every design decision should be reviewed by other developers as well as specialists in specific aspects of the application and equipment. Highly effective reviews can be based on the design documentation. If the documentation conforms to a standard structure, it can guide the review and make it less likely that important aspects of the decision will be overlooked [20].

3.3 Documentation based code inspections

A document specifying what that code should do is a prerequisite for a useful inspection of the actual code. If the code is complex, a “divide and conquer” approach, one that allows small units to be inspected independently of their context will be needed. A specification will be needed for each of those units. The specification for a unit, M, is used at least twice, once when inspecting M itself and again when inspecting units that use M. This is discussed in more depth in [21,22,23,26,29].

3.4 Documentation based revisions

Software that proves useful will, almost inevitably, be changed. Those who make the changes will not necessarily be the people who created the code and even if they are the same people they will have forgotten some details. They will need reliable information. Well-organized documentation can reduce costs and delays in the maintenance phase.

3.5 Documentation in contracts

An essential part of every well-written development contract is a specification that characterizes the set of acceptable deliverables. Specifications should not be confused with the contract. The contract also includes schedules, cost formulae, penalty clauses, statements about jurisdictions for dispute settlement, warranty terms, etc.

3.6 Documentation and attributing blame

Even in the best development projects, things can go wrong. Unless there is clear documentation of what each component must do, a 'finger-pointing' exercise, in which every group of developers blames another, will begin. With precise documentation, the obligations of each developer will be clear and blame (as well as costs) can be assigned to the responsible parties.

3.7 Documentation and compatibility

The chimera of interchangeable and reusable components is sought by most modern software development methods. Compatibility with a variety of components does not happen by accident; it requires a clear statement of the properties that the interchangeable components must share. This interface documentation must be precise; if an interface specification is ambiguous or unclear, software components are very unlikely to be interchangeable.

4 Costs and benefits of software documentation

Everyone is conscious of the cost of producing software documents because production costs are relatively easy to measure. It is much harder to measure the cost of *not* producing the documentation. This cost is buried in the cost of making changes, the cost of finding bugs, the cost of security flaws, the costs of production delays, etc. As a consequence it is hard to measure the benefits of good documentation. This section reviews some of those hidden costs and the associated benefits.

Losing time by adding people
In 1975, in a brilliant book of essays [2], Frederick P. Brooks, Jr. warned us that adding new staff to a late project could make it later. Newcomers need information that is often easily obtained by asking the original staff. During the of period when the newcomers are learning enough about the system to become useful, those who do understand the system spend much of their time explaining things to the newly added staff and, consequently, are less productive than before.

This problem can be ameliorated by giving the newcomers design documentation that is well-structured, precise, and complete. The newcomers can use this

documentation rather than interrupt the experienced staff. The documentation helps them to become useful more quickly.

Wasting time searching for answers
Anyone who has tried to change or debug a long program knows how long it can take to find key details in unstructured documentation or undocumented code. After the software has been deployed and is in the maintenance (corrections and updates) phase, those who must make the changes often spend more than half[4] of their time trying to figure out how the code was intended to work. Understanding the original design concept is a prerequisite to making safe changes. It is also essential for maintaining design integrity and efficiently identifying the parts of the software that must be changed.

Documentation that is structured so that information is easy to retrieve can save many frustrating and tiring hours.

Wasting time because of incorrect and inconsistent information
Development teams in a rush often correct code without making the corresponding changes in documentation. In other cases, with unstructured documentation, the documentation is updated in some places but not in others. This means that the documents become inconsistent and partially incorrect. Inconsistency and errors in documentation can lead programmers to waste time when making corrections and additions.

The cost of undetected errors
When documentation is incomplete and unstructured, the job of those assigned to review a design or to inspect the code is much harder. This often results in errors in releases and errors that are hard to diagnose. Errors in a released and distributed version of a product have many costs for both user and developer. Among other things, the reputation of the developer suffers.

Time wasted in inefficient and ineffective design reviews
During design reviews, much time is wasted while reviewers listen to introductory presentations before than can look for design errors. In fact, without meaningful design documents, design reviews often degenerate to a mixture of bragging session and tutorial. Such reviews are not effective at finding design errors. An alternative approach is active design reviews based on good documentation as described in [20].

Malicious exploitation of undocumented properties by others
Many of the security flaws that plague our popular software today are the result of a hacker discovering an undocumented property of a product and finding a way to use it. Often a security flaw escapes the attention of reviewers because they rely on the documentation and do not have the time to study the actual code. Many would be discovered and eliminated if there was better documentation.

[4]This is an estimate and is based on anecdotes.

5 Considering readers and writers

The first step towards producing better documentation is to think about who will read it, who will write it, what information the readers will need and what information the writers will have when they are writing.

Engineering documentation comprises many separate documents rather than a single, "all-in-one" document because there is a great variety of readers and writers. The readers do not all have the same needs or the same expertise. Each writer has different information to communicate. A developer's documentation standards must be designed with the characteristics of the expected readers and writers in mind.

Figure 1 lists a number of important software documents and the intended authors and audience for each.

Document	Writers	Readers/Users
Software Requirements Document	User reps, UI experts, application experts, controlled hardware experts	Authors of module guide and module interface specifications, (Software "Architects")
Module Guide	Software "Architects"	All Developers
Module Interface Specifications	Software "Architects"	Developers who implement or use the module
Program Uses Structure	Software "Architects"	Component Designers, Programmers
Module Implementation Design Document	Component Designers	Programmers implementing component
Display Method Program Documentation	Programmers implementing component	inspectors, maintainers potential reusers

Fig. 1. Software Document Readers and Writers

It is important to note that no two documents have the same readers or creators. For example, the people who have the information needed to determine the system requirements are not usually software developers. In an ideal world, representatives of the ultimate users would be the authors of requirements documentation. In today's world, developers might have to "ghost write" the document for those experts after eliciting information from them. This is far from ideal but it is a fact.

In contrast, determining the module structure of a product *is* a matter for software developers and, if the project has created a good[5] software requirements[6] document, the user representatives should not need to provide input or even review the module structure. Given a precise and complete module interface specification, the developers who implement the module need not ever consult the requirements documents, the

[5] The next section describes what is meant by "good".

[6] Sadly, most current requirements documents are not good enough to allow this. To do this, they would have to restrict the visible behaviour so precisely that any implementation that satisfied the requirements document would satisfy the users.

people who wrote those documents, or the user representatives. Unfortunately, the state of the art has not yet reached that nirvana.

The first step in producing good software documentation is to gain a clear picture of the authors and readers, their expertise and information needs.

The job titles that appear in the summary table (Figure 1) are buzzwords without precise definition. For example, "Software Architect" is almost meaningless. The ability to design interfaces is distinct from the ability to find a good internal structure for the software but a Software Architect may have either of those skills or both. Each development organization will have to produce its own version of Figure 1 based on its own job titles and associated skill descriptions.

With a set of readers identified, one must then ask how they will use a document, i.e. what questions they will use the document to answer. The document must be organized so that those questions will be easy to answer. Often several readers will have the same information needs but will ask different questions. For example, information about a component's interface will be needed by those who develop the component, those who test the component, those who use the component, and those assigned to maintain the component but they may be asking different questions. For example, a user may ask, "How do I get this information on the screen?", a tester or maintainer may ask, "When does this information appear on the screen?", a developer will be asking "What should the program display in these circumstances?". It is tempting to give each reader a separate document but this would be a mistake as they may get different information. The document should be structured so that any of those groups can answer their questions easily.

6 What makes design documentation good?

Above we have spoken of "good" design documentation. It is important to be precise about what properties we expect. The four most important are accuracy, lack of ambiguity, completeness, and ease of access.

6.1 Accuracy

The information in the document must be true of the product. Misinformation can be more damaging than missing information. Accuracy is most easily achieved if one can test that the product behaves as described. A precise document can be tested to see whether or not the product has the properties described in the document.

6.2 Unambiguous

The information must be precise, i.e. have only one possible interpretation. When reader and writer, (or two readers) can interpret a document differently, compatibility problems are very likely. Experience shows that it is very hard to write unambiguous documents in any natural language.

6.3 Completeness

As explained earlier, one should not expect any one document to describe everything about a product; each presents one view of the product. However, each view should be complete, i.e. all the information needed for that view should be given. For example, a black box description should describe the possible behaviour for all possible input sequences and a data structure description should include all data used by the product or component. A document format is good if it makes it easy to demonstrate completeness of a description.

6.4 Ease of access

With the complexity of modern software systems completeness is not enough. It must be possible to find that information quickly. This requires that clear organizational rules apply to the document. The rules should dictate where information is put in a document and can be used to retrieve information. There should be one and only one place for every piece of information.

7 Documents and mathematics

It is rare to hear anyone speak of software documentation and mathematics together. It is assumed that documents are strings in some natural language. In fact, documents are inherently mathematical and using mathematics in documentation is the only way to produce good documents.

7.1 Documents are predicates

We can examine a product, P, and a document to see if everything we can derive from the document is true of the product. There are 3 possibilities:

- The document is an accurate description of P.
- The document is not an accurate description of P.
- The document is meaningless or undefined when applied to P.[7]

In this way, a document partitions the set of products into three sets, those products for which it is *true*, those for which it is *false*, and those for which it is undefined. We can regard the document as a predicate; the domain of that predicate is the union of the first two sets. The third set comprises elements that are outside of the domain of P. If we wish to use the logic described in [25], we extend the predicate P to a total predicate P' by evaluating P' to *false* where P is not defined.

By treating documents as predicates, we can write "document expressions" to characterize classes of products.

[7] For example, a document that specified the colour of a product is meaningless for software products.

7.2 Mathematical definitions of document contents

Organizations that are serious about design documentation organize reviews for the documents that are produced. Many of the disagreements that arise during those reviews are about what information should be in the document, not about the design decisions that are made in the document. For example, some reviewers might argue that a data structure interface does not belong in a requirements document since it is a design decision. Others might suggest that this data will be communicated to other systems and the information about its representation is an essential part of the system requirements. Such disagreements are counterproductive as they distract from the real issues. Neither side would be considering the merit of the data structure itself. To minimize the time lost on such discussions, it is necessary to provide precise definitions of the content (not the format) of the documents.

A method of doing this is described in [24]. There it was shown that most design documents can be understood as representations of specific relations, for example the relation between input and output values. We will do this for a few key documents below.

7.3 Using mathematics in documents

To achieve the accuracy, lack of ambiguity, completeness, and ease of access that we expect of engineering design documents, natural language text should be avoided; the use of mathematics instead is essential. Once the contents of a document is defined abstractly as a set of relations, it is necessary to agree on how to represent those relations. A document such as the requirements document [4], described in [5], can consist of a set of mathematical expressions. These expressions represent the characteristic predicate of the relations being represented by the document. Representations that make these expressions easier to read, review, and write were used in [4] and are discussed and illustrated in section 9.

8 The most important software design documents

Each project will have its own documentation requirements but there is a small set of documents that is always needed. They are:

- The systems requirements document
- The module structure document
- Module interface documents
- Module internal design documents
- Program function documents

8.1 Requirements Documentation

The most fundamental obligation of a Professional Engineer is to make sure that their products are fit for use. This clearly implies that the Engineer must know what the requirements are. These requirements are not limited to the conscious wishes of the customer; they include other properties of which the customer might not be aware and requirements implied by the obligation of Engineers to protect the safety, well-being and property of the general public.

To fulfill their obligations, Engineers should insist on having a document that clearly states all requirements and is approved by the relevant parties. An Engineer must insist that this document is complete, consistent, and unambiguous.

No user visible decisions should be left to the programmers because programmers are experts in programming but often do not have expertise in an application or knowledge of the needs of users. Decisions about what services the user should receive should be made, documented, and reviewed by experts. The resulting document should constrain the program designers.

The two-variable model for requirements documentation

The two-variable model is a simple model that has been used in many areas of science and engineering. It is based on Figure 2.

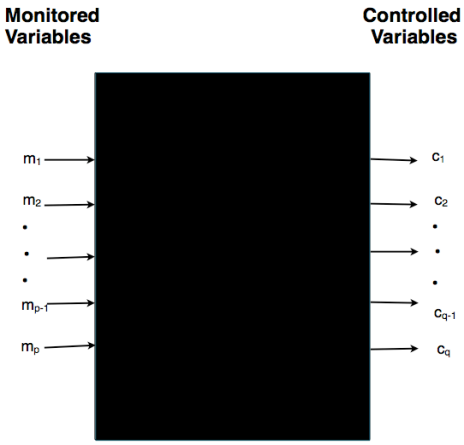

Fig. 2. Black Box View used for two-variable model.

A product can be viewed as a black box with p inputs and q outputs.

- We are given no information about the internals.
- The outputs, often referred to as *controlled variables*, $c_1,\ldots, c_q$, are variables whose value is determined by the system.
- The inputs, often known as *monitored variables*, $m_1,\ldots, m_p$, are variables whose value is determined externally.

- MC denotes $(m_1, \ldots m_p, c_1, \ldots c_q)$
- M(MC) and C(MC) denote $m_1, \ldots m_p$ and $c_1, \ldots, c_q$ respectively.
- If S is a set of MC-tuples M(S) is a set defined by $\{q \mid \exists T, T \ni S \wedge M(T) = q\}$ and C(S) is a set defined by $\{q \mid \exists T, T \ni S \wedge C(T) = q\}$.
- The values of a variable, v, over time can be described by a function of time denoted v^t. The expression $v^t(T)$ denotes the value of the v^t at time T.
- The values of any tuple of variables, V, over time can be described by a function of time denoted V^t. $V^t(T)$ denotes the value of the V^t at time T.

Viewing V^t as a set of ordered pairs of the form $(t, V^t(t))$:

- The subset of V^t with $t \leq T$ will be denoted V^t_T.

The product's possible behaviour can be described by a predicate, $SYSP(MC^t_T)$ where:

- MC^t_T is a history of the monitored and controlled values up to, and including, time T, and,
- $SYSP(MC^t_T)$ is *true* if and only if MC^t_T describes possible behaviour of the system.

This formulation allows the description of products such that output values can depend immediately on the input values (i.e., without delay). It also allows describing the values of the outputs by stating conditions that they must satisfy rather than by specifying the output as a function of the input[8].

It is sometimes convenient to use a relation, SYS derived from SYSP. The domain of SYS contains values of M^t_T and the range contains values of C^t_T.

- $SYS = \{(m, c) \mid \exists mc, SYSP(mc) \wedge C(mc) = c \wedge M(mc) = m\}$

In deterministic systems, the output values are a function of the input; consequently, the values of outputs in the history are redundant. In deterministic cases, we can treat SYS as a function with a domain comprising values of M^t_T and a range comprising values of C^t_T and to write $SYS(M^t_T)$ in expressions. $SYS(M^t_T)(T)$ evaluates to the value of the outputs at time T.

In the non-deterministic case, there are two complicating factors:

- The history need not determine a unique value for the output; SYS would not necessarily be a function.
- The output at time t may be constrained by previous output values, not just the input values.[9]

For these reasons, in the general (not necessarily deterministic) case the output values have to be included in history descriptions.

If the product is deterministic, SYS will be a function. In the non-deterministic case, the relation must either be used directly (e.g. as a predicate) or one can construct

[8] It is possible to describe conditions that cannot be satisfied. Care must be taken to avoid such nonsense.

[9] A simple example to illustrate this problem is the classic probability problem of drawing uniquely numbered balls from an opaque urn without replacing a ball after it is removed from the urn. The value that has been drawn cannot be drawn again, but except for that constraint, the output value is random.

a function whose range includes sets of possible output values rather than individual values.

For a 2-variable system requirements document we need two predicates NATP and REQP.

Relation NAT

The environment, i.e. the laws of nature and other installed systems, places constraints on the values of environmental quantities. They can be described by a predicate, NATP(MC^t_T), where:

- MC^t_T is a history of the input and output variable values up to, and including, time, T and,
- NATP(MC^t_T) if and only if MC^t_T describes possible behaviour of the product's environment if the product is not active in that environment.

It is sometimes convenient to represent NATP as a relation, NAT, with domain comprising values of MC^t_T and range comprising values of C^t_T. This is defined by:

- $NAT = \{(m, c) \mid \exists mc, NATP(mc) \wedge C(mc) = c \wedge M(mc) = m\}$

Relation REQ

No product can override NAT; it can only impose stricter constraints on the values of the output variables. The restrictions can be documented by a predicate, REQP(MC^t_T), where:

- MC^t_T is a history of the monitored and controlled variable values up to, and including, time T and,
- REQP(MC^t_T) is *true* if and only if MC^t_T describes permissible behaviour of the system.

It is sometimes convenient to represent REQP as a relation, REQ, with domain comprising values of MC^t_T and range comprising values of $C^t(T)$. This is defined by:

- $REQ = \{(m, c) \mid \exists mc, REQP(mc) \wedge C(mc) = c \wedge M(mc) = m\}$

If deterministic behaviour is required, we can write REQ (M^t_T), a function with values in the range, $C^t(T)$.

Experience and examples

Numerous requirements documents for critical systems have been written on the basis of this model. The earliest [4] are described in [5].

8.2 Software component interface documents

The two-variable model described above can be applied to software components, especially those that have been designed in accordance with the information hiding principle [13,14,18]. Because software components change state only at discrete points in time, a special case of the two-variable model, known as the Trace Function Method (TFM) can be used. TFM has been found useful for several industrial

products as described in [1,34,31,32]. These documents are easily used as referenced documents, can be checked for completeness and consistency, and can be input to simulators for evaluation of the design and testing of an implementation. The process of preparing these documents proved their usability; although they were developed by researchers, they were reviewed by practitioners who reported many detailed factual errors in the early drafts. If people cannot or will not read a document, they will not find faults in it.

8.3 Program function documents

Those who want to use a program do not usually want to know how it does its job; they want to know what job it does or is supposed to do. It has been observed by many mathematicians that a terminating deterministic program can be described by a function mapping from a starting state to a stopping state. Harlan Mills built an original approach to structured programming on this observation [11]. States were represented in terms of the values of program variables in those states. This was used extensively within IBM's Federal Systems Division. The main problem encountered was that developers found it difficult to represent the functions and often resorted to informal statements.

Other mathematicians have observed that one can describe non-deterministic programs with a relation from starting state to stopping states by adding a special element of the stopping state set to represent non-determination. This approach was perfectly general in theory but in practice, it proved difficult to represent the non-termination pseudo state in terms of program variable values.

A mathematically equivalent approach using two elements, a set of states in which termination was guaranteed and a relation was proposed by Parnas [17]. In this approach, both the set and the relation can be described in terms of program variable values making it easier to use the method in practice.

The remaining problem was the complexity of the expressions. This problem can be ameliorated by using tabular expressions (see below). The use of program function tables as program documentation has been illustrated in [27,26]. A further discussion of tabular expressions will be found below.

8.4 Module internal design documents

Many authors have noted that the design of a software component that has a hidden (internal) data structure can be defined by describing

- the hidden internal data structure,
- the program functions of each externally accessible program, i.e. their effect on the hidden data structure,
- an abstraction relation mapping between internal states and the externally distinguishable states of the objects created by the module.

The data structure can be described by programming language declarations. The functions are usually best represented using tabular expressions.

8.5 Documenting nondeterminism

One needs to be very careful when documenting programs that have non-deterministic behaviour.

We use the term "non-determinism" to describe situations where the output of a component is not-determined by the information available to us. For example, if a service can be provided by several different servers, each with satisfactory function but differences in performance, and we do not know which server we will get, we have non-determinism. If we had the information needed to predict which of the servers would be assigned to us, the behaviour would be deterministic.

Non-determinism is not the same as random behaviour. It is a matter of debate whether or not randomness exists but non-determinism exists without a doubt. The behaviour would be completely predictable by an observer that had the necessary information.

Non-determinism should not be confused with situations where there are alternatives that are equally acceptable but an acceptable alternative must be deterministic. For example, if purchasing a square-root finding program we might be happy with either a program that always gives the non-negative root or a program that always gives the negative root but we would not want one that alternated between the two.

Non-determinism can be documented by a relation. Choice as described in the previous paragraph requires a set of relations, one for each acceptable alternative.

When either non-determinism or choice is present it is often possible to write a using program that will get the desired result in any of the permitted cases. Documentation of the possible behaviour is essential for this.

Non-determinism is not an excuse for not providing precise documentation. The possible behaviours are always restricted and the set of possible characteristics must be characterized. In some cases there may be statistical information available and this too should be documented.

The same techniques that can be used for dealing with non-determinism can be used when developing software before some decisions about other programs or the environment have been made.

8.6 Additional documents

In addition to the system requirements document, which treats hardware and software as an integrated single unit, it is sometimes useful to write a software requirements document which is based on specific details about the input/output connections [24].

An informal document known as the module guide is also useful for larger systems. Hierarchically organized it classifies the modules by their "secrets" (the information hidden by each [18].

A uses relation document, which indicates which programs are used by each program is generally useful [16]. The information is a binary relation and may be represented in either tabular or graphical form.

In systems with concurrency, process structure documents are useful. The "gives work to" document is useful for deadlock prevention [15]. Interprocess/component communication should also be documented as discussed in [10].

9 Tabular expressions for documentation

Mathematical expressions that describe computer systems can become very complex, hard to write and hard to read. The power of software is its ability to implement functions that have many special cases. When these are described by expressions in conventional format, the depth of nesting of subexpressions gets to high. As first demonstrated in [5,4], the use of a tabular format for mathematical expressions can turn an unreadable symbol string into an easy to access complete and unambiguous document.

Figure 3 shows an expression that describes[10] the required behaviour of the keyboard checking program that was developed by Dell in Limerick, Ireland and described in [1,31].

Keyboard Checker: Conventional Expression

(N(T)=2∧keyOK∧(¬(T=_)∧N(p(T))=1))∨(N(T)=1∧(T=_∨(¬(T=_)∧N(p(T))=1))∧
(¬keyOK∧¬prevkeyOK∧¬prevkeyesc))∨((¬(T=_)∧N(p(T))=1)∧
((¬keyOK∧keyesc∧¬prevkeyesc)∨(¬keyOK∧keyesc∧prevkeyesc∧
prevexpkeyesc))∨((N(T)=N(p(T))+1)∧(¬(T=_)∧(1<N(p(T))<L))∧(keyOK))∨
((N(T)=N(p(T))-1))∧(¬keyOK∧¬keyesc∧(¬prevkeyOK∧prevkeyesc∧
preprevkeyOK)∨prevkeyOK)∧((¬(T=_)∧(1<N(p(T))<L))∨(¬(T=_)∧N(p(T))=L)))∨
((N(T)=N(p(T)))∧(¬(T=_)∧(1<N(p(T))≤L))∧((¬keyOK∧¬keyesc∧(¬prevkeyOK∧
prevkeyesc∧¬preprevkeyOK))∨(¬keyOK ∧¬prevkeyOK∧ ¬prevkeyesc)∨
(¬keyOK∧keyesc∧¬prevkeyesc)∨(¬keyOK∧keyesc∧prevkeyesc∧
prevexpkeyesc))∨((N(P(T)=Fail)∧(¬keyOK∧keyesc∧prevkeyesc∧
¬prevexpkeyesc)∧(1≤N(p(T))≤L))∨((N(P(T)=Pass)∧(¬(T= _)∧N(p(T))=L)∧(keyOK))

Fig. 3. Characteristic Predicate of a Keyboard Checker Program

Figure 4 is a tabular version of the same expression. It consists of 3 grids. For this type of expression, the top grid and left grid are called "headers". The lower right grid is the main grid. To use this type of tabular expression one uses each of the headers to

[10] Auxiliary predicates such as keyesc, keyOK, etc. are defined separately. Each is simply defined.

select a row and column. If properly constructed, only one of the column headers and one of the row headers will evaluate to *true* for any assignment of values to the variables. The selected row and column identify one cell in the main grid. Evaluating the expression in that cell will yield the value of the function for the assigned values of the variables. The tabular expression parses the conventional expression for the user. Instead of trying to evaluate or understand the complex expression in Figure 3, one looks at the simpler expressions in Figure 4. Generally, one will only need to evaluate a proper subset of those expressions.

N(T) =

			T=_	¬ (T= _) ∧		
				N(p(T))=1	1<N(p(T))< L	N(p(T))= L
keyOK				2	N(p(T))+ 1	Pass
¬keyOK ∧	¬keyesc ∧	(¬prevkeyOK ∧ prevkeyesc ∧ preprevkeyOK) ∨ prevkeyOK			N(p(T)) - 1	N(p(T)) - 1
		¬prevkeyOK ∧ prevkeyesc ∧ ¬preprevkeyOK			N(p(T))	N(p(T))
		¬prevkeyOK ∧ ¬ prevkeyesc	1	1	N(p(T))	N(p(T))
	keyesc ∧	¬ prevkeyesc		1	N(p(T))	N(p(T))
		prevkeyesc ∧ ¬prevexpkeyesc		Fail	Fail	Fail
		prevkeyesc ∧ prevexpkeyesc		1	N(p(T))	N(p(T))

Fig. 4. Tabular expression equivalent to Figure 3.

There are many forms of tabular expressions. The grids need not be rectangular. A variety of types of tabular expressions are illustrated and defined in [9]. Although tabular expressions were successfully used without proper definition in a number of applications, precise semantics are needed for tools and full analysis. There have been four basic approaches to defining the meaning of these expressions. Janicki and his co-authors developed an approach based on information flow graphs that can be used to define a number of expressions [8]. Zucker based his definition on predicate logic. Khédri and his colleagues based their approach on relational algebra [3]. All three of these approaches were limited to certain forms of tables [35]. The most recent approach [9], which is less restricted, defines the meaning of these expressions by means of translation schema that will convert any expression of a given type to an equivalent conventional expression. This is the most general approach and provides a good basis for tools. The appropriate table form will depend on the characteristics of the function being described. Jin shows a broad variety of useful table types in [9] which provides a general approach to defining the meaning of any new type of table.

10 Summary and Outlook

This paper has argued that it is important to the future of software engineering to learn how to replace the poorly structured, vague and wordy design documentation that we use today with precise professional design documents. It has also shown that documents have a mathematical meaning and are composed of mathematical expressions; those expressions can be in a variety of tabular formats that have proven to be practical over a period of more than 30 years. While there is much room for improvement, and a variety of questions for researchers to answer, what we have today can, and should, be used. Unfortunately, it is rarely taught. Nonetheless successful applications in such areas as flight-software, nuclear power plants, manufacturing systems, and telephone systems show that we are at a point where it is possible and practical to produce better documentation this way.

From its inception, the software development has been plagued by a mentality that enjoys and glorifies the excitement of "cut and try" software development. Some programmers talk with a gleam in their eyes about the excitement of trying new heuristics, the exciting challenge of finding bugs. Many of them talk as if it is inherent in software that there will be many bugs and glitches and that we will never get it right. One said, "The great thing about this is that there is always more to do." One gets the impression that the view software development as a modern version of the old pinball machines. When you win, bells ring but when you lose there is just a dull thud and a chance to try again.

Over the same period most software developers have been quite disparaging about documentation and the suggestion that it should conform to any standards. I have been told, "You can criticize my code but I have the right to write my documentation the way I like", and, "If I had wanted to document, I would have been an English major". In fact, one of the current fads in the field (which has always been plagued by fashions described by buzzwords) advises people not to write any documentation but just focus on code.

It is time to recognize that we have become a serious industry that produces critical products. We can no longer view it as a fun game. We need to grow up. The first step towards maturity must be to take documentation seriously, and to use documentation as a design medium. When our documentation improves in quality, the software quality will improve too.

References

1. Baber, R., Parnas, D.L., Vilkomir, S., Harrison, P., O'Connor, T.: Disciplined Methods of Software Specifications: A Case Study. In: Proceedings of the International Conference on Information Technology Coding and Computing (ITCC'05). IEEE Computer Society (2005)
2. Brooks, F.P. Jr.: The Mythical Man-Month: Essays on Software Engineering, 2nd Edition. Addison Wesley, Reading, MA (1995)
3. Desharnais, J., Khédri, R., Mili A.: Towards a Uniform relational semantics for tabular expressions. In: Proceedings of RelMiCS'98, pp. 53-57 (1998)

4. Heninger, K., Kallander, J., Parnas, D.L., Shore, J.,: Software Requirements for the A-7E Aircraft. Naval Research Laboratory Report 3876 (1978)
5. Heninger, K.L.: Specifying Software Requirements for Complex Systems: New Techniques and their Application. IEEE Transactions Software Engineering, vol. SE-6, pp. 2-13 (1980)
   - Reprinted as chapter 6 in [7]
6. Hester, S.D., Parnas, D.L., Utter, D.F.: Using Documentation as a Software Design Medium, Bell System Technical Journal 60(8), pp. 1941-1977 (1981)
7. Hoffman D.M., Weiss, D.M. (eds.): Software Fundamentals: Collected Papers by David L. Parnas. Addison-Wesley (2001)
8. Janicki, R.: Towards a formal semantics of Parnas tables. In: Proc. of 17th International Conference on Software Engineering (ICSE), pp. 231-240 (1995)
9. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. Science of Computer Programming 75(11), pp. 980-1000 (2010)
10. Liu, Z., Parnas, D.L., Trancón y Widemann, B.: Documenting and Verifying Systems Assembled from Components. Frontiers of Computer Science in China, Higher Education Press, co-published with Springer-Verlag, in press (2010)
11. Mills, Harlan D.: The New Math of Computer Programming. Communications of the ACM 18(1): 43-48 (1975)
12. Meyer, B.: Eiffel: The Language. Prentice-Hall (1991)
13. Parnas, D.L.: Information Distributions Aspects of Design Methodology. In: Proceedings of IFIP Congress '71, Booklet TA-3, pp. 26-30 (1971)
14. Parnas D.L.: On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15(12), pp. 1053-1058 (1972)
   - Translated into Japanese - BIT, vol. 14, no. 3, 1982, pp. 54-60.
   - Republished in Classics in Software Engineering, edited by Edward Nash Yourdon, Yourdon Press, 1979, pp. 141-150.
   - Republished in Great Papers in Computer Science, edited by Phillip Laplante, West Publishing Co, Minneapolis/St. Paul 1996, pp. 433-441.
   - Reprinted as Chapter 7 in [7]
   - Reprinted in Software Pioneers: Contributions to Software Engineering, Manfred Broy and Ernst Denert (Eds.), Springer Verlag, pp. 481 - 498 (2002)
15. Parnas, D.L.: On a 'Buzzword': Hierarchical Structure, IFIP Congress '74, North Holland Publishing Company, pp. 336-339 (1974)
   - Reprinted as Chapter 8 in [7]
   - Reprinted in Software Pioneers: Contributions to Software Engineering, Manfred Broy and Ernst Denert (Eds.), Springer Verlag, pp. 501 - 513 (2002)
16. Parnas, D.L.: Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering, pp. 128-138 (1979)
   - Also in Proceedings of the Third International Conference on Software Engineering, pp. 264-277 (1978)
   - Reprinted as Chapter 14 in [7]
17. Parnas, D.L.: A Generalized Control Structure and its Formal Definition. Communications of the ACM 26(8), pp. 572-581 (1983)
18. Parnas, D.L., Clements, P.C., Weiss, D.M.: The Modular Structure of Complex Systems. IEEE Transactions on Software Engineering, vol. SE-11 no. 3, pp. 259-266 (1985)
19. Parnas, D.L., Wadge, W.: A Final Comment Regarding An Alternative Control Structure and its Formal Definition (Technical Correspondence). Communications of the ACM 27(5), pp. 499-522 (1984)
20. Parnas, D.L., Weiss, D.M.: Active Design Reviews: Principles and Practices. In: Proceedings of the 8th International Conference on Software Engineering (1985)
   - Also published in Journal of Systems and Software, December 1987.

- Reprinted as Chapter 17 in [7].
21. Parnas, D.L., Asmis, G.J.K., Kendall, J.D.: Reviewable Development of Safety Critical Software. The Institution of Nuclear Engineers, International Conference on Control & Instrumentation in Nuclear Installations, paper no. 4:2 (1990)
22. Parnas, D.L., van Schouwen, A.J., Kwan, S.P.: Evaluation of Safety-Critical Software. Communications of the ACM 33(6), pp. 636-648 (1990)
    - Published in Advances in Real-Time Systems, John A. Stankovic and Krithi Ramamritham (editors), IEEE Computer Society Press, pp. 34-46 (1993)
    - Published in Ethics and Computing: Living Responsibly in a Computerized World, Kevin W. Bowyer (editor), IEEE Press, pp. 187-199 (2000)
23. Parnas D.L., Asmis, G.J.K., Madey, J.: Assessment of Safety-Critical Software in Nuclear Power Plants. Nuclear Safety 32(2), pp. 189-198 (1991)
24. Parnas, D.L., Madey, J.: Functional Documentation for Computer Systems Engineering. Science of Computer Programming 25(1), pp. 41-61 (1995)
25. Parnas D.L.: Predicate Logic for Software Engineering. IEEE Transactions on Software Engineering 19(9), pp. 856-862 (1993)
26. Parnas, D.L: Inspection of Safety Critical Software using Function Tables. In: Proceedings of IFIP World Congress 1994, Volume III, pp. 270 - 277 (1994)
    - Reprinted as Chapter 19 in [7]
27. Parnas, D.L., Madey, J., Iglewski, M.: Precise Documentation of Well-Structured Programs. IEEE Transactions on Software Engineering 20(12), pp. 948-976 (1994)
28. Parnas, D.L.: Teaching Programming as Engineering. In: The Z Formal Specification Notation, 9th International Conference of Z Users (ZUM'95), Lecture Notes in Computer Science vol. 967, Springer-Verlag, pp. 471-481 (1995)
    - Reprinted as Chapter 31 in [7]
29. Parnas, D.L.: Using Mathematical Models in the Inspection of Critical Software. In: Applications of Formal Methods. Prentice Hall International Series in Computer Science, pp. 17-31 (1995)
30. Parnas, D.L.: Structured programming: A minor part of software engineering. In: Special Issue to Honour Professor W.M. Turski's Contribution to Computing Science on the Occasion of his 65th Birthday, Information Processing Letters 88(1-2), pp. 53-58 (2003)
31. Parnas, D.L., Vilkomir, S.A: Precise Documentation of Critical Software. In: Proceedings of the Tenth IEEE Symposium on High Assurance Systems Engineering (HASE'07), pp. 237-244 (2007)
32. Parnas, D. L.: Document Based Rational Software Development. Knowledge-Based Systems 22(3), pp. 132-141 (2009)
33. Smith, B.: http://www.microsoft.com/presspass/press/2006/jan06/01-25EUSourceCodePR.mspx
34. Quinn, C., Vilkomir, S.A., Parnas, D.L. Kostic, S.: Specification of Software Component Requirements Using the Trace Function Method. In: Proceedings of the International Conference on Software Engineering Advances (ICSEA'06) (2006)
35. Zucker, J.I.: Transformations of normal and inverted function tables. Formal Aspects of Computing 8, pp. 679-705 (1996)