

## Software Development: What is Still Missing?

### David Lorge Parnas

#### Abstract

The proposal that software development should be a profession modeled on the engineering disciplines was widely discussed in the 1960s. At that time, most Computer Scientists had been trained as physical scientists or mathematicians. Engineers developed the hardware but left the software development to others (computer users). Many of the scientists and mathematicians who were developing software realized that their job was more like engineering than science. They had been taught to create and organize knowledge, but software development was producing products that others would use. This talk “takes stock” of the progress since that time and identifies what remains to be done.

Computers today can perform services that were unimaginable in the ‘60s. However, many problems remain. Software products commonly have a number of “bugs” and other problems that we would not expect or accept in a physical product. A close look shows that many of the notable advances are made possible by improvements in hardware, not improvements in the way we construct software.

Three things that are considered essential for a mature profession are missing in software development are:

- rigid entrance standards for the profession
- education that prepares students to meet those standards
- professional documentation standards similar to those used in other engineering disciplines.

The least discussed of these three topics is documentation. It is viewed as a nontechnical skill that is not a suitable subject for research for Computer Scientists. The talk shows how we can use structured mathematical notation to provide precise documentation that is demonstrably complete, and unusually useful to developers, reviewers, and maintainers. It then makes some suggestions for turning software development into a mature engineering profession.

1.	Progress in Four Decades-----	3	34.	Requirements Documentation-----	36
2.	Doubts -----	4	35.	The two-variable model for requirements documentation-----	37
3.	Lessons From Dijkstra's work -----	5	36.	the Two-Variable Model is Not Appropriate for Software -----	38
4.	What About Education for Software Engineers? -----	6	37.	The four-Variable model for requirements documentation -----	39
5.	More on Education -----	7	38.	Nondeterminism -----	40
6.	Licensing: Another Gap -----	8	39.	Experience and examples: Requirements -----	41
7.	Recording and Communicating Design Details -----	9	40.	Does it Scale? Does it work in the Real World" -----	42
8.	Documentation: a perpetually unpopular topic -----	10	41.	Interfaces -----	43
9.	The Words of an Experienced Developer -----	11	42.	Surprising Observations about Interfaces. -----	44
10.	Dilbert Knows that documentation is important. -----	12	43.	Software component interface documents -----	45
11.	Programming vs. software Engineering -----	13	44.	Part I of Clock Interface Document-----	46
12.	What Is Meant by "Document" In Engineering -----	14	45.	Part II of Clock Interface Document-----	47
13.	A Preliminary Example: Dell Keyboard Checker -----	15	46.	Extract from Module Interface Document-----	48
14.	Auxiliary functions defined on 2nd Page -----	16	47.	Program function documents -----	49
15.	The remainder of the Keyboard Checker Document -----	17	48.	Example of Program-Function Table -----	50
16.	Tabular Expressions: There is No Theoretical Advantage! -----	18	49.	Program-Function for a Poor (real) Program -----	51
17.	Why It Is Important to Call this a Document -----	19	50.	Subtables for Nuclear Plant Code -----	52
18.	Documentation as An Information Retrieval Problem -----	20	51.	Module internal design documents -----	53
19.	Completeness and Consistency -----	21	52.	Checking an Internal Design -----	54
20.	Are computer programs self-documenting? -----	22	53.	Additional documents -----	55
21.	Internal documentation vs. separate documents -----	23	54.	Tabular expressions for documentation -----	56
22.	Models vs. documents -----	24	55.	There are many forms of tabular expressions. -----	57
23.	Design documents vs. introductory documentation -----	25	56.	Tables like this Can be Found on the Internet-----	58
24.	Specifications vs. other descriptions (1) -----	26	57.	This says the same thing -----	59
25.	Specifications vs. other descriptions (2) -----	27	58.	This Too is A Mathematical Expression-----	60
26.	Extracted documents -----	28	59.	A Circular Table-----	61
27.	Documents Are Not Programs -----	29	60.	Is My proposal Different from "Formal Methods"? -----	62
28.	Roles played by documents in development - 1 -----	30	61.	The Bottom Lines: -----	63
29.	Roles played by documents in development - 2 -----	31	62.	Management's Role in Document Driven Design -----	64
30.	Costs and benefits of software documentation -----	32	63.	Research Problems -----	65
31.	The most important software design documents -----	33	64.	Summary and Outlook -----	66
32.	Considering readers and writers -----	34	65.	Real Improvement is Difficult -----	67
33.	Documents and mathematics -----	35			

## Progress In Four Decades

### User Interfaces

- Early conference paper: *Display Use for Man-Machine Dialog* (eds. W. Händler, J. Weizembaum)
- Today's interfaces unimaginable then! (Bürgermeister: “In my dream, I talked with my toy train.”)

### Parallel Processing common

### Better treatment of variability, product lines, etc. (not “real problem” in 1969)

### Automation of update process (Assembly once a major barrier!)

### “Almost” standard interfaces for networks.

### Languages that **try** to reflect structuring principles

### Examples: Lessons of the T.H.E system - Applicable in all current systems

- Uses Hierarchy
- Multithreading (called processes)
- Deadlock prevention
- Stepwise refinement

Unfortunately, you won't find this in many products.

## Doubts

### User Interfaces: How much of the advance is actually hardware?

- My early paper: “*Sample Man Machine Interface Specification-A Graphics Based Line Editor*”
- Implementation attempt was stupid: Windows were not implementable at that time.

### Parallel processing common

- Research that once started with Illiac has started again (from zero). Problems still unsolved.
- Dijkstra approach never discussed.

### Better treatment of variability, product lines, etc.

- Many unnecessary differences between products of one company. (e.g. Apple, Nokia)
- Retrofitting rather than upfront design of families. (e.g Apple)

### Automation of update process

- One-way street with potholes
- Not module replacement

### “Almost” standard interfaces for networks. “Almost” says it all! Its a euphemism for “not”.

### Languages that **try** to reflect structuring principles

- Force many implementation decisions on users. Not always the appropriate decisions.

## Lessons From Dijkstra's Work

Many of them still applicable!

- Structured Programming
  - Principle of refinement often ignored
  - Little understanding of why “go to” considered harmful
- Separation of Concerns (when used, it is applied in an *ad hoc* way)
- Hiding the number of processors
- Transput streams (pipes)
- Predicate transformers (There are both better and worse ways but wp still useful.)

These are principles - not technology!

Many modern graduates know only the technology.

Leading researchers for whom the ideas should be accessible believe that anything old is no longer valuable. They pass this belief on to students.

## What About Education For Software Engineers?

As an EE, I could talk and work with students from anywhere (even MIT).

As a software specialist, I always find communication difficult.

- I don't know what they know and don't know or what I might not know.
- Every school seems to have different terminology and notation.
- Professors teach their favorite topics and neglect things they consider dull or trivial. I often ask someone where they studied so I can communicate with them.

We have yet to agree on a Core Body of Knowledge.

We do not consistently follow the “professional education” model.

- Teach both theory and practice!
- Teach how to apply theory in practice. (CS programmes keep them separate.)
- Distinguish between current technology and “eternal” principles.
- Teach appreciation for real standards, disciplined processes, reviews, etc.
- Teach professional responsibility in a meaningful way.

## More On Education

200. Parnas, D.L., “Software Engineering Programmes are not Computer Science Programmes”, *Annals of Software Engineering*, vol. 6, 1998, pgs. 19-37.

- Reprinted (by request) in *IEEE Software*, November/December 1999, pp. 19-30.

### Abstract

Programmes in “Software Engineering” have become a source of contention in many universities. Computer Science departments, many of which have used that phrase to describe individual courses for decades, claim software engineering as part of their discipline. Some engineering faculties claim “Software Engineering” as a new speciality in the family of engineering disciplines. We discuss the differences between traditional computer science programmes and most engineering programmes and argues that we need software engineering programmes that follow the traditional engineering approach to professional education. One such programme is described.

## Licensing: Another Gap

If you have created a web page, you can call yourself a “Software Engineer”.

Why not?

- What definition or rule would you be violating?
- Who would tell you that you could not use that title?
- What evidence could anyone demand of you?

This is not the case in Law, Medicine, Engineering, or Hair Cutting!

- Those are organized professions! Software Engineering is not yet one of them.

Do we agree on what Software Systems Engineers must know?

- Should they know what a loop invariant is and how to use it?
- Should they know how to check for termination of a loop?
- Should they understand how to design abstract interfaces?
- Should they understand how to design a product line as a program family?
- Should they know how to use the 4-variable requirements model?

Experts can, and do, disagree on these questions. We have work to do.

## Recording And Communicating Design Details

Bridge designers can communicate precisely with design documentation

This is true of automobile designers, aircraft designers, chemical manufacturers, etc., etc.

They have documentation standards that allow interchange (sometimes dictated by government regulations).

We have no such thing.

This is something where researchers can help.

## Documentation: A Perpetually Unpopular Topic

Software documentation is disliked by almost everyone.

- Program developers don't want to prepare documentation.
- User documentation is often left to technical writers who do not necessarily know all the details. Their documents are often initially incorrect, inconsistent and incomplete.
- The intended readers find the documentation to be poorly organized, poorly prepared and unreliable; they do not want to use it. Most prefer "try it and see" or "look at the code" to looking in documentation. ("Always wrong!")
- User documentation is often displaced by "help" systems because it is hard to find the details that are sought in conventional documentation. Unfortunately, the "help" system only answers a set of frequently occurring questions; it is usually incomplete and redundant. Those with an unusual question don't get much help.
- Computer Science researchers do not see software documentation as a research topic. They can see no mathematics, no algorithms, etc.

These factors feed each other in a vicious cycle.

Bad documentation is not used and, therefore, does not get improved.

## The Words Of An Experienced Developer

"Documentation means the tedious task of reading thru a finished code, and making a Doc/pdf file which is used during Project Reviews or during resolution of blame-games as it inevitably happens at some point of time. This document is never used by the next programmer. Generally the Software is again modified (ported to another platform, or significantly changed because the rules of the world have changed) much later, when nobody understands the document any more. As an example, when I started my work (there were no MSWord/Acrobat in 1972), I got about 300 loose sheets left by a previous developer with flow charts and Explanations of flow charts. I left it in my drawer unread until I finished the compiler on my own.

So, Document means something that we all hate with our heart."

(Basudeb Gupta, Private Communication)

The word are a true reflection of the current situation in industry!

Can we do something?

## Dilbert Knows That Documentation Is Important.



When people leave, knowledge leaves with them.

## Programming Vs. Software Engineering

“Software Engineering” is not just another name for programming.

Programming is only a small part of software Engineering.

- “Software” refers to “*a program or set of programs written by one group of people for repeated use by another group of people*”<sup>1</sup>. This is fundamentally different from producing a program for a single use, or for your own use.
- When producing a program for your own use, you can expect the user to understand the program and to know how to use it. There is no need to prepare manuals, to explain what parameters mean, to specify the format of the input, etc. All of these things are required when preparing a program that will be used by strangers.
- When producing a program for a single use, there is no need to design a program that can be easily maintained in several versions (product line) and no need to describe the design decisions to those who will have to change it.

These differences between software development and programming (multi-person involvement, multi-version use) make documentation essential for software development.

- You can be a good programmer and a bad software developer, but
- You cannot be a good software developer and a bad programmer.

<sup>1</sup> Brian Randell was the first to point this out to me.

## What Is Meant By “Document” In Engineering

A record of design decisions that is binding for developers.

- Documents restrict future decisions.
- Deviations require an approved change.

To be as useful as possible documents must be:

- Accurate
- Consistent
- Complete (all decisions fully documented).
- Explicitly structured for easy retrieval and easy change.

Informal introductions/explanations are not documents in this sense.

Documents are not written afterwards; they are the design medium.

Vague documents are like vague contracts<sup>2</sup>; worse than having none at all.

- Agile: for communication but against documentation?

<sup>2</sup> A design document is an essential part of a contract but not the whole contract.

## A Preliminary Example: Dell Keyboard Checker

In daily use in Limerick Ireland for many years.

Claimed to be completely correct.

Two informal descriptions totaling (only) 21 pages (English).

- several ambiguities
- a few missing cases
- a few errors

Challenge by a skeptical manager, “Do better!”.

All information could be expressed in two pages.

- Preparation of those pages revealed errors in both program and older documents
- New document much more precise and easily used.
- New document suitable as input to testing tools and inspection process.

## Keyboard Checker: Tabular Expression

$N(T) =$

			$\neg (T = \_) \wedge$		
			$N(p(T)) = 1$	$1 < N(p(T)) < L$	$N(p(T)) = L$
$\neg \text{keyOK} \wedge$	$\neg \text{keyesc} \wedge$	keyOK	2	$N(p(T)) + 1$	Pass
		$(\neg \text{prevkeyOK} \wedge \text{prevkeyesc} \wedge \text{preprevkeyOK}) \vee \text{prevkeyOK}$		$N(p(T)) - 1$	$N(p(T)) - 1$
		$\neg \text{prevkeyOK} \wedge \text{prevkeyesc} \wedge \neg \text{preprevkeyOK}$		$N(p(T))$	$N(p(T))$
	keyesc $\wedge$	$\neg \text{prevkeyOK} \wedge \neg \text{prevkeyesc}$	1	$N(p(T))$	$N(p(T))$
		$\neg \text{prevkeyesc}$	1	$N(p(T))$	$N(p(T))$
		$\text{prevkeyesc} \wedge \neg \text{prevexpkeyesc}$	Fail	Fail	Fail
		$\text{prevkeyesc} \wedge \text{prevexpkeyesc}$	1	$N(p(T))$	$N(p(T))$

## Auxiliary Functions Defined On 2<sup>nd</sup> Page<sup>3</sup>

<sup>3</sup> Not the latest version of this method! NOTATION CAN BE SIMPLIFIED BUT STRUCTURE DOES NOT CHANGE.

## The Remainder Of The Keyboard Checker Document

Name	Meaning	Definition
keyOK	most recent key is the expected one	$r(T) = N(p(T))$
keyesc	most recent key is the escape key	$r(T) = \text{esc}$
prevkeyOK	key before the most recent key was expected one	$r(p(T)) = N(p(p(T)))$
prevkeyesc	key before the most recent key was escape key	$r(p(T)) = \text{esc}$
preprevkeyOK	key 2 keys before most recent key was expected key	$r(p(p(T))) = N(p(p(p(T))))$
prevexpkeyesc	key expected before most recent key was escape key	$N(p(p(T))) = \text{esc}$

We have found a way to simplify this notation

We can train both developers and managers to read these documents.

We have trained some to write such documents.

Experience in reading eases learning to write.

## Tabular Expressions: There Is No Theoretical Advantage!

The table is a mathematical expression, mathematically equivalent to the one below.

Keyboard Checker: Conventional Expression
$  \begin{aligned}  & (N(T)=2 \wedge \text{keyOK} \wedge (\neg(T=_) \wedge N(p(T))=1)) \vee (N(T)=1 \wedge (T=_) \vee (\neg(T=_) \wedge N(p(T))=1)) \wedge \\  & (\neg \text{keyOK} \wedge \neg \text{prevkeyOK} \wedge \neg \text{prevkeyesc}) \vee ((\neg(T=_) \wedge N(p(T))=1) \wedge \\  & ((\neg \text{keyOK} \wedge \text{keyesc} \wedge \neg \text{prevkeyesc}) \vee (\neg \text{keyOK} \wedge \text{keyesc} \wedge \text{prevkeyesc} \wedge \\  & \text{prevexpkeyesc})) \vee ((N(T)=N(p(T))+1) \wedge (\neg(T=_) \wedge (1 < N(p(T)) < L)) \wedge (\text{keyOK})) \vee \\  & ((N(T)=N(p(T))-1)) \wedge (\neg \text{keyOK} \wedge \neg \text{keyesc} \wedge (\neg \text{prevkeyOK} \wedge \text{prevkeyesc} \wedge \\  & \text{preprevkeyOK}) \vee \text{prevkeyOK} \wedge ((\neg(T=_) \wedge (1 < N(p(T)) < L)) \vee (\neg(T=_) \wedge N(p(T))=L))) \vee \\  & ((N(T)=N(p(T))) \wedge (\neg(T=_) \wedge (1 < N(p(T)) \leq L)) \wedge ((\neg \text{keyOK} \wedge \neg \text{keyesc} \wedge (\neg \text{prevkeyOK} \wedge \\  & \text{prevkeyesc} \wedge \neg \text{preprevkeyOK})) \vee (\neg \text{keyOK} \wedge \neg \text{prevkeyOK} \wedge \neg \text{prevkeyesc}) \vee \\  & (\neg \text{keyOK} \wedge \text{keyesc} \wedge \neg \text{prevkeyesc}) \vee (\neg \text{keyOK} \wedge \text{keyesc} \wedge \text{prevkeyesc} \wedge \\  & \text{prevexpkeyesc})) \vee ((N(P(T))=\text{Fail}) \wedge (\neg \text{keyOK} \wedge \text{keyesc} \wedge \text{prevkeyesc} \wedge \\  & \neg \text{prevexpkeyesc}) \wedge (1 \leq N(p(T)) \leq L)) \vee ((N(P(T))=\text{Pass}) \wedge (\neg(T=_) \wedge N(p(T))=L) \wedge (\text{keyOK}))  \end{aligned}  $

The advantages are practical, not theoretical.

- Ease of reference
- Checkability
- Fewer errors

## Why It Is Important To Call This A Document

Industry recognizes the need for documentation, but

- Time pressure often causes them to postpone it or not do it at all.
- They do not know how to do it better.
- **What they produce is not very useful; it often sits unused.**
- **Developers find the code easier to use and more trustworthy.**

The inadequacy of today's documentation resulted in "agile" methods.

Developers will not have time to prepare both current and better documents.

They have to see precise, structured documentation as an improvement on what they do and not as an addition to what they already do.

The purpose of this documentation is communication, not proof!

- It is designed for information retrieval.
- It is designed for ease of checking for completeness and consistency.

## Documentation As An Information Retrieval Problem

Design documents are places to put information so that:

- Reviewers can get the information they need.
- Developers can get the information they need.
- Testers can get the information they need.
- Modifiers (maintainers) can get the information they need.

The key to information retrieval is having strict rules for:

- what information to store,
- where to store that information, and
- how to store information.

The same rules can then be used to retrieve information.

## Completeness And Consistency

Documentation is expected to be complete and consistent, but...

- Individual documents are never complete descriptions of a system.
- They are only complete relative to a document content specification.
- The complete set of documents should form a complete description.
- There should be minimal duplication of information.
- Unresolved Issues or missing information must be explicitly noted.
- Each document gives a different view of the software.

Information accompanying today's software often comes with disclaimers, i.e., statements that deny any claim to accuracy.

- This is not a property of engineering documents.
- Part of professional responsibility is taking responsibility for documents.

## Are Computer Programs Self-documenting?

Code itself looks like 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*". (response to EU fines)

No such confusion with physical products; there is a clear distinction

- between a circuit diagram and the circuit
- between a bridge and its blueprints.

Code is commonly described as self documenting

- This may be true "in theory" but, in practice, it is a naive illusion or disingenuous. It is very hard to use.
- We need documents that contain the essential (binding) information, abstracting from the huge amounts of information in the code that we do not need.
- We should be able to use a program without reading its code.
- Code does not distinguish between required, incidental, and unintended properties.

## Internal Documentation Vs. Separate Documents

Nobody wants documentation distributed within a physical product.

- Nobody wants to climb a bridge to determine the sizes of nuts and bolts
- Drivers do not want to look at the bridge structure to know load limits.

We expect the documentation to be separate from the product.

Some propose that assertions, or program functions, and similar information be placed in the code. (e.g. Bertrand Meyer)

- This is useful to the developers and maintainers but not other readers.
  - Testers should be able to prepare "black box" test suites before code completion.
  - Programmers using a program should not have to read it.
  - People should be able to use a program without understanding its code.

## Models Vs. Documents

Renewed interest in models and "model-driven engineering."

There is an important distinction between "model" and "document".

**Definition:** A *model* of a product is a simplified depiction of that product; a model may be either physical (usually reduced in size and detail) or abstract.

- A model will have some important properties of the original.
- Not all properties of the model are properties of the actual system.

**Definition:** A *mathematical model* of a system is a mathematical description of the properties of a model of that product.

- Mathematical models can be very useful to developers but, because they are not necessarily accurate descriptions; they may not be suitable as documents.
- One can derive information from some models that is not true of the real system.
- Consequently, models must be used with great care;
- Every precise and accurate document can constitute a safe mathematical model Everything you can derive from it will be true of the real thing.



## Design Documents Vs. Introductory Documentation

When we write something, 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 allow a reader to retrieve specific facts.
- Tutorials are intended for 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.

Compare introductory language textbooks with dictionaries.

- Textbooks begin with the easier and more fundamental aspects of the language.
- Dictionaries arrange words in a specified order that is not based on the above.
- Narratives make poor reference works
- Reference works are a poor way to get an introduction to a subject.

We need both kinds of documents but this talk is about reference documents.

## Specifications Vs. Other Descriptions (1)

We must be conscious of the role that a document will play in a development process. There are two basic roles, description and specification.

- Descriptions provide properties of a product that exists (or once existed).
- Specifications are descriptions that state only the required properties of a product.
- A specification that states all required properties is called a full specification.
- Descriptions may include properties that are incidental and not requirements.
- If a product does not satisfy a specification, it is not acceptable for the use intended.

The difference is one of intent, not form or even content.

- Every specification that a product satisfies is also a description of that product.
- The notation can be the same.
- This has confused many researchers.
- **There is no such thing as a “specification language”.**

## Specifications Vs. Other Descriptions (2)

Distinction is important when one product is used as a component of another.

- The builder of the using product may assume that any replacements will still have the properties stated in a specification.
- This is not true if the document is a description that is not a specification.
- Users should not rely on descriptions that are not specifications.

Specifications impose obligations on implementers, purchasers, and users.

- 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. (no “best effort”)
- Users must be able to count on the properties stated in a specification;
- Users must not base their work on any properties not stated in the specification.
- Purchasers are obligated to accept, and pay for, a product that meets the (full) specification included in a purchase agreement or bid.

## Extracted Documents

It is possible to produce a description by examining the product.

- Extracted documents will be descriptions but not usually specifications.
- Observation or inspection cannot tell you what was intended or what is required.
- Extracted documents usually contain low-level information, not abstractions.
- Extracted documentation is of little value during development.
- Extracted documents not a valid guide for testers. Would be circular; You are assuming that the code is correct and testing to see that it does what it does.
- Documentation based on comments can be untrustworthy.

Javadoc like tools are of very limited use.

- Used by developers who do not want to document. (lazy)
- Depend on comments
- Are unable to distinguish between incidental and required properties

## Documents Are Not Programs

They describe mappings from input to output without describing the steps in the computation process or any other information that should not be in the document<sup>4</sup>.

They must provide the exactly the information that the intended readership needs in a way that is easy for them to use.

Our documents are mathematical expressions describing a function that maps an input to an output.

Documents must answer questions put by users, i.e. “If this happens, what might the output be”.

<sup>4</sup> Content definitions for documents will be discussed later.

## Roles Played By Documents In Development - 1

### Documentation as the design medium

- Decisions are made by putting them in documents.

### Documentation-based design reviews

- Creating documentation reveals problems
- Reviewing documentation is an early design review.

### Documentation based code inspections

- Reviewing programs against their specification
- Reviewing the programs that use that program.
- Divide and conquer using hierarchical decomposition and displays

### Documentation based revisions

- Maintainers need guidance
- Developers may have forgotten, quit, died, become managers.....

## Roles Played By Documents In Development - 2

### Documentation in contracts

- Specification of the set of acceptable deliverables is an essential part of contracts.
- Contract also includes schedules, cost formulae, penalty clauses, statements about jurisdictions for dispute settlement, warranty terms, etc.

### Documentation is used to attribute blame and settle disputes

- Who did not conform?
- Which component is wrong?

### Documentation and compatibility

- The chimera of interchangeable and reusable components will not be achieved without a clear precise specification for those components.

### Documentation as a medium for the parties to communicate.

- Volker Gruhn's observations: “Communication the key determiner of success.”
- Communication both when writing documents and when they are used.
- Documentation implements structured communication both when writing and afterwards.

### Documentation is the key to distributed development.

## Costs And Benefits Of Software Documentation

Documentation production costs seem easy to measure.

Much harder to measure the cost of not producing the documentation.

What matters is the net cost - production cost minus savings.

Losing time by adding people.

- Frederick P. Brooks, Jr.: Adding new staff to a late project can make it later.
- Newcomers need information Experienced staff become less productive
- Good documentation ameliorates the problem.

Time is wasted searching for answers.

- Documentation that is structured for information retrieval saves frustrating hours.

Time is wasted because of incorrect and inconsistent information

The cost of detecting errors late or never is higher than early detection.

Time is wasted in inefficient and ineffective design reviews.

Malicious exploitation of undocumented properties by hackers.



## The Most Important Software Design Documents

Each project will have its own documentation requirements.

There is a small set of documents that is always needed. They are:

- The Systems Requirements document
- The Module Structure document (module guide, informal)
- Module interface documents
- Module internal design documents
- Program function documents

## Considering Readers And Writers

Many separate documents because of variety of readers and writers

The readers have different needs; Writers have different information

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 <u>implement</u> or <u>use</u> 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

No two documents have the same readers or creators.

## Documents And Mathematics

It is rare to speak of software documentation and mathematics together.

Documents are predicates.

- We can write "document expressions" to characterize classes of products.

Mathematical definitions of document contents are needed.

- Avoid the endless discussions about, "What goes where?"
- Avoid duplication and missing information.

Using mathematics in documents

- Necessary for accuracy, lack of ambiguity, completeness, and ease of access
- The contents of a document can be defined abstractly as a set of relations
- Representation of this information is a critical issue.
- If it cannot be read, it is not a useful document.

Engineers use mathematics. Technicians might not.

## Requirements Documentation

Professional Engineers must make sure that their products are fit for use.

- This implies that an Engineer must know what the requirements are.

An Engineer need not determine requirements, but must check them.

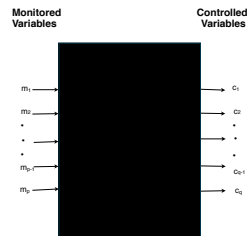
- Requirements are not limited to the conscious wishes of the customer.
- Other requirements implied by the obligation of Engineers to protect the safety, well-being and property of the general public.

Engineers should insist on having a complete, consistent, and unambiguous document that has been approved by all relevant parties..

**No user visible decisions should be left to the Engineers/programmers**

## The Two-variable Model For Requirements Documentation

The two-variable model has been used in many areas of engineering.



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

- We are given no information about the internals.
- Values of controlled variables,  $c_1, \dots, c_q$ , are determined by the system.
- Values of monitored variables,  $m_1, \dots, m_p$ , are determined externally.
- Output values can depend immediately on the input values (i.e., without delay)

## The Two Variable Model Is Not Appropriate For Software

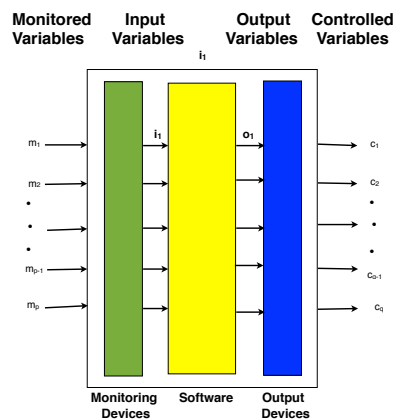
Two-variable model applied to software (yellow box)

Relations are complex and not meaningful to users.

The input and output devices transform the information about the monitored variables in complex ways. The relation between the inputs to the software and its outputs can be too complex.

Users usually know what the monitored and controlled variables mean but not the inputs and outputs.

## The Four-Variable Model For Requirements Documentation



Looking Inside the Black Box to distinguish Hardware from Software.

## Nondeterminism

In deterministic systems, the output values are a function of the input.

- The values of outputs in the history are redundant.
- We can treat SYS as a function: domain: values of  $M^T$ , range: values of  $C^T$ .
- $SYS(M^T)(T)$  evaluates to the value of the outputs at time  $T$ .

In the nondeterministic case, there are two complicating factors:

- Relation SYS would not necessarily be a function.
- The output may be constrained by previous output values, not just the input values.<sup>5</sup>

In the general case output values must be included in history descriptions.

For a 2-variable system requirements document we need two predicates NATP and REQ. These are discussed next.

<sup>5</sup> 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.

## Experience And Examples: Requirements

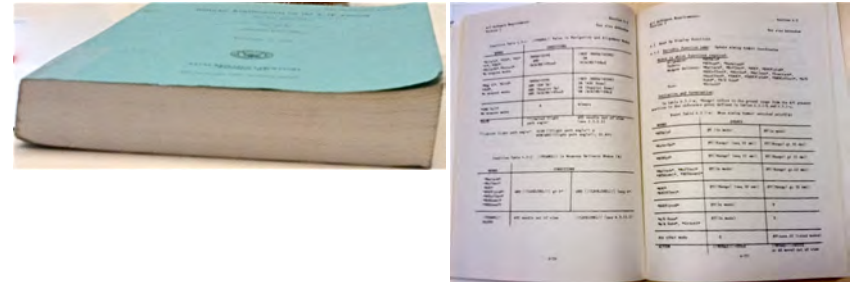
Numerous requirements documents written using this model.

- A-7 OFP [HKPS] [Heninger].
  - Pilots found hundreds of detail errors
  - Programmers coded from document
- Bell Laboratories SES
  - Copied by others
  - "Shortest soak time"
- Darlington Nuclear Power Generating Plant
  - basis for a successful inspection
- Dell keyboard checker
  - Found errors in existing documents, 21 pages reduced to 2

Continued by NRL/SCR.

We could do better today and better yet tomorrow. (room for research)

## Does It Scale? Does It Work In The Real World?



Answer any behaviour question by looking at no more than 7 pages.

Pilots found more than 500 detail errors in draft versions.

Format copied for other versions of aircraft and other aircraft.

Used to improve quality assurance, maintenance, contracting.

## Interfaces

One of the most important, and least well understood, concepts in Software Engineering.

Often, confused with syntax of invocations or a shared data structure.

### Definition: Interface

*Given two communicating software components, A and B, B's interface to A is the weakest assumption about B that would allow you to prove that A is correct.*

Any change in B that invalidates its interface to A means that, A could not be proven correct and should be changed.

Interfaces determine the difficulty of changing software.

Interface documents allow independent development.

## Surprising Observations About Interfaces.

- There isn't necessarily a 1:1 relation between a program and an interface.
- Interfaces not symmetric. B's interface to A differs from A's interface to B.
- B may have an interface to A even if A does not have an interface to B.
- A component may have a specified interface. This tells the developers of other programs what they may assume about the specified component.
- If the developers of a component, A, make use of facts about a specified component, B, that are not implied by B's specified interface, the actual interface is stronger than the specified interface and A should be considered incorrect (even if it is working).
- B may have an interface with A even if neither invokes the other. For example, the correctness of A may depend on B maintaining a shared data structure with certain properties.
- Published interface (assumption that can be made by all) should imply the actual pairwise interfaces but sometimes does not (bad error).

Software Component Interface Documents

Two-variable model can be applied to software components  
Discrete event version of the two-variable model, known as the Trace Function Method (TFM), can be used.

- TFM documents are
- easily used as reference documents,
  - can be checked for completeness and consistency
  - can be input to simulators for evaluation of the design and testing an implementation.
  - can be reviewed by practitioners who reported many detailed factual errors

If people cannot read a document, they will not find faults in it.

Part I of Clock Interface Document

min(T) ≡

PGM(r(T)) = SET HR		min(p(T))
PGM(r(T)) = SET MIN ∧	0 ≤ 'in(r(T)) ≤ 59	'in(r(T))
	¬ (0 ≤ 'in(r(T)) ≤ 59)	min(p(T))
PGM(r(T)) = INC ∧	min(p(T)) = 59	0
	¬ (min(p(T))=59)	min(p(T)) + 1
PGM(r(T)) = DEC ∧	¬ (min(p(T))= 0)	min((p(T)))-1
	min(p(T))= 0	59
T= _		0

Part II Of Clock Interface Document

$hr(T) \equiv$

$PGM(r(T)) = SET \text{ HR} \wedge$		$0 \leq \text{'in}(r(T)) < 24$	$\text{'in}(r(T))$
		$\neg (0 \leq \text{in}(r(T)) < 24)$	$hr((p(T)))$
$PGM(r(T)) = SET \text{ MIN}$			
			$hr((p(T)))$
$PGM(r(T)) = INC \wedge$	$\min(p(T)) = 59 \wedge$	$hr(p(T)) = 23$	0
		$\neg hr(p(T)) = 23$	$1 + hr((p(T)))$
	$\neg (\min(p(T)) = 59)$		$hr((p(T)))$
$PGM(r(T)) = DEC \wedge$	$\neg (\min(p(T)) = 0)$		$hr((p(T)))$
	$\min(p(T)) = 0 \wedge$	$\neg (hr(p(T))) = 0$	$hr((p(T))) - 1$
		$hr(p(T)) = 0$	23
$T = \_$			0

Extract From Module Interface Document

Output Functions

hr(T) ≡

PGM(r(T)) = SET HR ∧		0 ≤ 'in(r(T)) < 24	'in(r(T))
		¬ (0 ≤ in(r(T)) < 24)	hr((p(T)))
PGM(r(T)) = SET MIN			hr((p(T)))
PGM(r(T)) = INC ∧	min(p(T))= 59 ∧	hr(p(T))= 23	0
		¬ hr(p(T))= 23	1+ hr((p(T)))
	¬ (min(p(T))=59)		hr((p(T)))
PGM(r(T)) = DEC ∧	¬ (min(p(T))= 0)		hr((p(T)))
	min(p(T))= 0 ∧	¬ (hr(p(T)))= 0	hr((p(T)))-1
		hr(p(T))= 0	23
T= _			0

## Program Function Documents

Those who use a program need not know how it works.

- They want to know what it does or is supposed to do.

Terminating deterministic program can be described by a function mapping from a starting state to a stopping state.

States represented in terms of the values of program variables.

Theoretically, non-deterministic programs can be described a relation from starting state to stopping states plus a special element for non-determination.

In practice, LD-relation (relation plus termination set) is better.

- Allows all formulae to be in terms of actual program variables.

Mathematically equivalent but better in practice.

Tabular expressions make it work in practice.

**Big** programs, when well-written, have **small** tables.

## Example Of Program-Function Table

Test	external variables: e, V, index, found, low, high, med		
$R_3(.) = ('low \leq 'med \leq 'high) \Rightarrow$			
	'V['med]		
	< 'e	= 'e	> 'e
index' =	true	index' = 'med	true
found' =	'found	true	'found
low' =	'med + 1	'low	'low
high' =	'high	'high	'med - 1

$\wedge NC(e, V, med)$

## Program-Function For A Poor (real) Program

	'OKTTI = FALSE.	('OKTTI = TRUE.) AND NOT 'NoSensTrip!	('OKTTI = TRUE.) AND . 'NoSensTrip!
B('PTBI,'IDOW1I)	B('PTBI,'IDOW1I .OR. 'FMASK('PTBI)#)	Table 4	B('PTBI,'IDOW1I .OR. 'FMASK('PTBI)#)
B('CN#,'IDOW2I)	B('CN#,'IDOW2I)	Table 4	B('CN#,'IDOW2I)
B('CND#,'IDOW2I)	B('CND#,'IDOW2I)	Table 4	B('CND#,'IDOW2I)
'HEXI	'HEXI .OR. 'IMASKI	'HEXI .OR. 'IMASKI	'HEXI .OR. 'IMASKI
'HI1I	'HI1I	'/HTL(S)/ - 'HYSI	'/HTL(S)/ - 'HYSI
'HI2I	'HI2I	'/HTL(S)/	'/HTL(S)/
'ILO1I	'ILO1I	'/LTL(S)/	'/LTL(S)/
'ILO2I	'ILO2I	'/LTL(S)/ - 'HYSI	'/LTL(S)/ - 'HYSI
'IMCI	'IMCI	Table 4	0
'IPCI	'IPCI	Table 4	0
B('JSTBVI), j = 'ISTBI + j - 1, i in {1..5}	B('JSTBVI)	Table 3	Table 3
B('JSTBVI), NOT (j in ('ISTBI + i - 1), i in {1..5})	B('JSTBVI) .AND. 'IUMI)	B('JSTBVI) .AND. 'IUMI)	B('JSTBVI) .AND. 'IUMI)
B('ISTBI + i - 1, 'ISTWII .OR. 'IUMI)	B('ISTBI + i - 1, 'ISTWII .OR. 'IUMI)	Table 3	Table 3
B('JSTWII), NOT (j in ('ISTBI + i - 1), 'ISTWII .OR. 'IUMI)	B('JSTWII) .OR. 'IUMI)	B('JSTWII)	B('JSTWII)
B('TTBI,'TTWII)	B('TTBI,'TTWII .OR. 'FMASK('TTBI)#)	B('TTBI,'TTWII .AND. 'FMASK('TTBI)#)	B('TTBI,'TTWII .AND. 'FMASK('TTBI)#)
'IHIF(1..5I)	'IHIF(1..5I)	Table 2	Table 2
'III	'III	6	6
'ILOF(1..5I)	'ILOF(1..5I)	Table 2	Table 2

## Subtables For Nuclear Plant Code

Table 2

	'ABvHiHys(I)	'InHiHys(I)	'InNorm(I)	'InLoHys(I)	'BlwLoHys(I)
'IHIF(I)I	.FALSE.	'IHIF(I)I	.TRUE.	.TRUE.	.TRUE.
'ILOF(I)I	.TRUE.	.TRUE.	.TRUE.	'ILOF(I)I	.FALSE.

Table 3

$A^* = [ ('IIMCI \geq 'IDELI) \text{ OR } ('IIMCI < 0) \text{ OR } ('IPCI + 1 \geq 'IPCI) \text{ OR } (('IIMCI + 1) < 0)]$

	'IPCI	'IDELI	'IIMCI + 1
'IIMCI	'IDELI	'IIMCI	'IIMCI
B('IPTBI,'IDOW1I)	B('IPTBI,'IDOW1I .AND. 'FMASK('IPTBI)#)	B('IPTBI,'IDOW1I)	B('IPTBI,'IDOW1I)
B('CN#,'IDOW2I)	B('CN#,'IDOW2I .AND. 'FMASK('CN#)#)	B('CN#,'IDOW2I)	B('CN#,'IDOW2I)
B('CND#,'IDOW2I)	B('CND#,'IDOW2I .AND. 'FMASK('CND#)#)	B('CND#,'IDOW2I)	B('CND#,'IDOW2I)

Table 4

$A^* = [ ('IIMCI \geq 'IDELI) \text{ OR } ('IIMCI < 0) \text{ OR } ('IPCI + 1 \geq 'IPCI) \text{ OR } (('IIMCI + 1) < 0)]$

	'IPCI	'IDELI	'IIMCI + 1
'IIMCI	'IDELI	'IIMCI	'IIMCI
B('IPTBI,'IDOW1I)	B('IPTBI,'IDOW1I .AND. 'FMASK('IPTBI)#)	B('IPTBI,'IDOW1I)	B('IPTBI,'IDOW1I)
B('CN#,'IDOW2I)	B('CN#,'IDOW2I .AND. 'FMASK('CN#)#)	B('CN#,'IDOW2I)	B('CN#,'IDOW2I)
B('CND#,'IDOW2I)	B('CND#,'IDOW2I .AND. 'FMASK('CND#)#)	B('CND#,'IDOW2I)	B('CND#,'IDOW2I)

## Module Internal Design Documents

Design of a software component is documented by describing: (Many authors)

- 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.

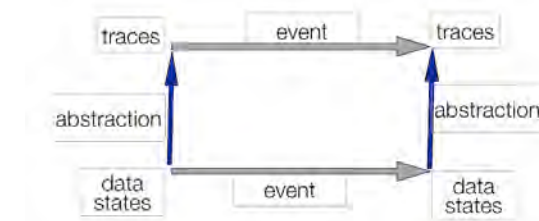
The functions are usually best represented using tabular expressions.

Easily extended to non-deterministic case using relations.

## Checking An Internal Design

Design documentation should allow us to verify the workability of a design.

For all possible events,  $e$ , the following must hold:



$$AR(d1, t1) \wedge e(d1, d2) = AR(d2, t1.e)$$

The information is there for an informal check.

No examples yet.

## 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
- An informal document known as the module guide
- A uses relation document, which indicates which programs are used by each program is generally useful. 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.
  - Interprocess/component communication should also be documented

## Tabular Expressions For Documentation

Mathematical expressions that describe computer systems can become very complex, hard to write and hard to read.

As first demonstrated in 1977, the use of a tabular format for mathematical expressions can turn an unreadable symbol string into an easy to access complete and unambiguous document.

Pilots were able to find 500 errors in our first draft.



### There Are Many Forms Of Tabular Expressions.

- The grids need not be rectangular.
- A variety of types of tabular expressions are illustrated and defined [Jin].
- [Jin], defines the meaning of these expressions by means of translation schema to an equivalent conventional expression.
- Good basis for tools.
- The appropriate table form will depend on the characteristics of the function being described.

### Tables Like This Can Be Found On The Internet

Ticket Price	1 Passenger	2 Passengers	3-5 Passengers	6 or more Passengers
0 - 100	35	45	55	65
101 - 200	40	50	60	70
201 - 300	45	55	65	75
301 and more	50	60	70	70 + 10 per passenger

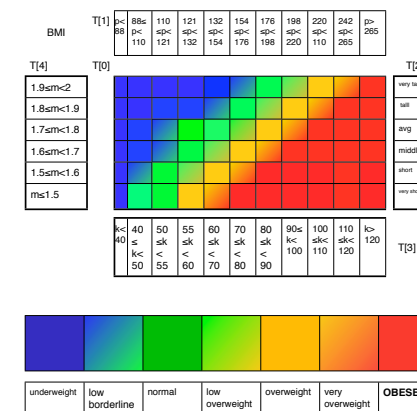
Such tables are familiar and intuitive.

### This Says The Same Thing

	P=1	P=2	2 < P < 6	P > 5
0 < T ≤ 100	35	45	55	65
100 < T ≤ 200	40	50	60	70
200 < T ≤ 300	45	55	65	75
T > 300	50	60	70	70 + 10 × (P-5)

However, the above is a mathematical expression.

### This Too Is A Mathematical Expression



### A Circular Table



### Is My Proposal Different From “Formal Methods”?

It is no less formal. In fact, it is arguably more formal.

However, there are important differences:

- Intended for documentation, not proof or models
- Careful attention to document content (readers and writers)
- Designed for use as a reference document
- Concern for readers and writers and their needs leads to structured documentation.
- Developed in practice, formalized later
- Evolved from practical experience, strengthened through theory
- Engineering mathematics, not philosophers/logicians mathematics
- Mathematics is general, not tailored to program description.

The phrase “formal methods” was a mistake. Engineers always use mathematics; developers who do not are not Engineers.

It is not just the tables that make it different.

### The Bottom Lines:

Producing no documentation gets developers in trouble.

Producing bad documentation might be worse.

Producing good documentation:

- helps them to get the requirements right
- helps them to get interfaces right
- helps them to in their testing
- helps them to in their inspections
- helps them in maintenance and upgrades
- helps them manage a product line effectively.

Define the content of each document (as illustrated)

Use appropriate (mathematical) tabular expressions

### Management's Role In Document Driven Design

Management is getting something done without knowing exactly what it is (and much more).

Management can undermine any effort by either not demanding it, not leaving time for it, or not supporting it.

- Insist that if it isn't documented, it is not done.
- Schedule document reviews
- Insist that software testers test against documents using the documents to generate oracles and test cases.
- Insist on document guided inspections for critical parts.
- Allow no change without revising the associated documents.

Without management support it won't work!

### Research Problems

More documents (e.g. sequential process structure)  
Various forms of composition given these documents  
Reliability given these documents  
More table types  
More examples (real and publishable)  
Improved notation  
Tools that are more than Masters theses

### Summary And Outlook

It is important to the future of software engineering to learn how to replace today's documentation with precise professional design documents.  
Documents must have a mathematical meaning.  
The expressions can be in a tabular formats.  
These have proven to be practical over a period of more than 30 years.  
There is much room for improvement and research is needed.  
No more "cut and try" software development.  
Software has become a serious industry that produces critical products.  
The first step towards maturity must be to take documentation seriously  
When our documentation improves, the software quality will improve too.

### Real Improvement Is Difficult

"Nobody" does it that way.

- "Nobody" builds really good software (error free, easily maintained)

We don't have time to write documents that nobody reads.

- "Never have time at the start, always have time at the end" (B.O. Evans via F.P. Brooks)

"I have no idea how to do that" (Ph.D. developer, author)

- Nobody taught you how! Nobody is teaching how to document software.

Dilbert's view on making real changes:



- Ideas that would change the way we work can be very threatening.