Design through Documentation: The Path to Software Quality

David Lorge Parnas, P.Eng, Ph.D, Dr.h.c., Dr.h.c., FRSC, FACM, FCAE

SFI Fellow, Professor of Software Engineering
Director of the Software Quality Research Laboratory (SQRL)
Department of Computer Science and Information Systems
Faculty of Informatics and Electronics
University of Limerick

Abstract

In traditional engineering design, preparation of a sequence of documents precedes the actual construction begins. Each document is used for review and analysis and, after revision, serves as input to the next phase in the development. When errors are discovered or changes are required, the design documents previously approved are updated and reviewed again. Each new document is reviewed against the previous documents. Whenever a document is revised, those based on it are reviewed and revised if necessary.

In software design this approach is rarely properly applied. Practitioners seem unable or willing to write the precise documents that would be required. Instead, they write vague statements that cannot be subject to rigorous analysis and are of little value to those making the next decisions.

We will provide precise definitions of a set of software documents and how these documents can be produced as part of an improved software development process.

Topics

Why we need precise documentation - motivation

The documents that we need - content definitions.

How to do it - illustrations.

The taste of success and future plans (experience and dreams)

Software Components: the Elusive Dream

Software should be a collection of software components.

- Nobody can build products as one big “blob”.
- Everyone wants to re-use software components.
- “Components are junk!” (industry leader)
- “We have invested millions in writing reusable components that are never reused”.

What goes wrong?

- Components are often based on assumptions that do not hold in new environment.
- Components have complex interfaces that make them hard to use.
- Components are not well documented - making them hard to use.

Why documentation is the key.

- Requiring documentation, encourages better design - simpler interfaces
- Requiring documentation often reveals hidden assumptions (exposing them)
- Documentation makes the component useable.

If it ain’t documented it ain’t done.

Why is Software So Often a Problem?

Developers *consistently* underestimate the difficulty of building software for long-term use.

They *write* software rather than *design* it.

They do not:

- systematically, identify and record requirements,
- hold reviews of the requirements document,
- explicitly design, document and review software structure,
- carefully inspect all designs and programs.

These steps are standard practice for all engineering products other than software.

The steps are not taken for software because,

- "Software is easy!"
- ***"The code is self-documenting!"***
- "Software is *just* a set of instructions."
- "Anyone who knows the language can program."

Famous last words!

Why do we need complete documentation?

Whenever we discover a software problem of any kind, we realise that there was something that we did not understand.

If we understood our software, it would be more secure, more reliable, more trustworthy.

If we had complete documentation, we would have a better understanding of our software because:

- Good documentation helps us to understand our software.
- You cannot produce good documentation without understanding the software.

Why do we need precise documentation?

In software, “almost right” means “wrong”.

- Little things mean a lot
- Security violations come from exploiting small mistakes.

In software, “vague” means “easily misinterpreted”.

- “Remove the top element of the stack” might mean “set it to zero”.
- “address” might mean “physical address” or “IP address”.

When something is “almost right” it might work most of the time but

- fail at unexpected times, or
- be exploited by someone with criminal intent.

Why do we need abstract documentation?

Abstract does not mean imprecise or vague!

- We need precise documentation.

Abstract does not mean “wrong”.

- An unbounded buffer is not an abstraction, it is a lie.

Abstract means that, in a particular document, we omit certain details.

- The details that are ignored must be documented elsewhere.
- The details that are ignored must be irrelevant for some purpose.

Unless we have precise abstract documentation, we will not understand our software. It will be inscrutable.

A Simple Example: String Copy in Unix-like Systems.

Security vulnerability often results from exploitation of unexpected behaviour in routines that copy strings. Do you:

- know what your string copy routine does if the string to be copied is longer/shorter than it should be?
- know what your string copy routine does if the source string is longer than the destination location?

You need to know those things to avoid vulnerability.

Do you

- know if a successful copy is performed from left to right or right to left?
- know the order in which preconditions are checked?

You do not need to know those things to avoid vulnerability.

Documentation must be precise about the things that we need to know and abstract from the things that are irrelevant for the reader's purpose.

The documentation should be easier to read than the code.

Why Don't People Apply Engineering Discipline to Software?

1. Some don't do not understand the meaning of "engineering discipline".
2. Some don't think it's necessary because the market doesn't demand it.
3. Some don't know how to do it.

In this talk I want to focus on how to do it but we must also understand the importance of doing it.

Is Documentation Relevant?

"We have better things to do than document"

"We sell code, not design documents."

But,

- We cannot collect, review, or check requirements for completeness unless we document them.
- We can't make, review, or live up to structural decisions unless they are documented.
- We can't inspect designs, without design documentation.
- We can best inspect programs with the help of program documentation.
- Some maintenance programmers estimate that 80% of their time is spent seeking information in documents and verifying its accuracy.

Design through documentation is key to better software.

Two Aspects of Better Software:

(1) Better design

(2) Better documentation

Two Aspects of Better Documentation

(1) Better design (easier to document)

(2) Using mathematics, which is

- more compact,
- less ambiguous,
- more useful (mechanically interpretable)

than natural language.

Two Aspects of Better Design

(1) Following good software design principles

(2) Raising consciousness: documenting design and making decisions more explicit and reviewable.

In other words, design and documentation are irrevocably linked. They help (or hurt) each other.

The Documents that Software Dependant Organisations Need

1. A **Requirements Document** that tells the users exactly what they will get and tells the programmers exactly what to build.
   - The challenge is to use one document that is useful to both groups.
2. A **Module-guide** that guides reviewers and maintenance-programmers to exactly the programs that will be affected by a proposed change.
   - For all who work on the project, it is restricted in the designs to which it applies
3. A set of **Module Interface Specifications** that tell programmers what they must build and other programmers, what they can expect of a module.
   - This is the only information read by both those who build the module and those who use it.
4. A set of **Module Internal Design Documents** that (1) record major internal design decisions for reviewers, (2) guide the programmers in their coding, (3) help maintainers to understand the code.
   - This information is only for developers/maintainers of the module in question.

So why don't we do it?

The fundamental answer, "*We do not know how*".

Imagine trying to develop an automobile if you do not know how to specify the thread on nuts and bolts.

Imagine trying to develop electronic devices and systems if we do not know how to specify impedance, voltage, current, inductance, frequency response.

Imagine trying to produce for the petroleum industry if you do not know how to specify viscosity, octane level, etc.

Imagine trying to design electronic circuits without a knowledge of circuit theory.

You do not have to imagine trying to produce software systems without knowing how to specify component properties. *We do it every day.*

Never Build on Sand

Software is among the most complex of products.

We cannot hope to keep such products under intellectual control, unless we build on a solid, sound, well understood foundation.

If

- we don't know the meaning of our notation, or
- we don't know the associated rules of inference, or if
- those rules are complex and conditional,

using that notation will not help us to understand what we are doing.

To build complex products all design notations must have a sound and simple mathematical basis.

Today, many people promote Undefined Modelling Languages (UMLs) as a solution. Without a precise definition of their meaning, UMLs become part of the problem, not the solution.

Retrofit of definitions, never works. Once people start using vaguely defined notations, the definitions proliferate like weeds.

Divide and Conquer for Requirements Documentation

Consider the controlled variables one at a time.

- Describe each one as a function of monitored variables.
- Both monitored and controlled variables may be added during this process.

Identify modes of operation.

- Describe transitions to define modes (using monitored variables)
- Define conditions within each mode

For each of the identified controlled variables:

1. Prepare an empty table with rows for each mode and columns for each condition.
2. Complete the table with expressions using only monitored variables.
3. Monitored and controlled variables will be added during the design process.
   - the process reveals forgotten variables
   - internal variables may be monitored and displayed for diagnostics.

Bringing Time into the Picture

All of these variables can vary with time.

For each scalar variable, x, denote the time-function describing its value by "$x^{\mathbf{t}}$".

The value of x at time t is denoted "$x^{\mathbf{t}}$ (t)".

The vector of time-functions ($v^{\mathbf{t}}_{\mathbf{1}}$, $v^{\mathbf{t}}_{\mathbf{2}}$, ..., $v^{\mathbf{t}}_{\mathbf{n}}$) will be denoted by "$\underline{v}^{\mathbf{t}}$".

Contrary to the statements by some computer scientists, there is no problem dealing with "real" time. No new mathematics is needed.

For many systems, the time functions may be replaced by functions of event sequences.

Mathematical Definition of Document Contents

The implementors need to know the following relations:

Relation NAT:

- domain contains values of $\underline{m}^t$, range contains values of $\underline{c}^t$,
- ($\underline{\mathbf{m}}^{\mathbf{t}}$, $\underline{c}^t$) is in NAT if and only if nature permits that behaviour.

This tell us what we need to know about the environment.

Relation REQ:

- domain contains values of $\underline{\mathbf{m}}^{\mathbf{t}}$, range contains values of $\underline{c}^t$,
- ($\underline{\mathbf{m}}^{\mathbf{t}}$, $\underline{c}^t$) is in REQ if and only if system should permit that behaviour.

This tells us how the new system is intended to further *restrict* what NAT(ure) allows to happen. (Within NAT's domain, REQ must be a subset of NAT)

If we can describe these relations, we have documented the system requirements.

We can get the "scary" math out of the documents by using the right notation.

What is the result of this process?

Requirements documents that are complete and precise.

Requirements documents that can be reviewed (and errors found) by users (such as pilots, telephone operators) and engineers who know nothing about software.

Requirements documents that are precise enough to allow programs to be written without further discussion.

Requirements documents that can be used to generate test programs[1].

Requirements documents that pass the “coffee stain” test. They serve as the communications medium between those who know the application area and those who know the software technology.

The labour is “front-end investment” that is repaid in coding and revision.

The approach strikes most people as counter-intuitive until they do it.

---

[1] If mathematical notation is used.

Why Use This Approach?

(1) For all the “motherhood” reasons that we try to find the requirements first.
(2) Because we can check for completeness.
(3) Because we can check for consistency.
(4) Because we have a precise description.
(5) Because we have a reviewable document.
(6) Because we can often simulate the system.
(7) Because the design can be based on the document.
(8) Because the programming goes much faster.
(9) Because the programmers work consistently and do not duplicate each other’s work.
(10) Because we will discover ways to simplify the system.
(11) Because we can build monitors for testing or supervising the system.

Why not?

(1) Because it requires some training.
(2) Because it is a *front-end* investment that slows down the *initial* part of development.

How can Documents be both Precise and Readable? (I)

This is precise:

$$(((\exists\, i,\ B[i] = \text{`x}) \wedge (B[j'] = x) \wedge (present' = \textbf{true})) \vee ((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \neq x)) \wedge (present' = \textbf{false}))) \wedge (\text{`x = x'} \wedge \text{`B = B'})$$

But,

- Few people want to read even simple examples like this.
- You have to parse it all and understand a lot before you can find what you want.

Lets try anyway - just once.

$$(((\exists\, i,\ B[i] = \text{`x}) \wedge (B[j'] = x) \wedge (present' = \textbf{true})) \quad \vee ((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \neq x)) \wedge (present' = \textbf{false}))) \bigwedge (\text{`x = x'} \wedge \text{`B = B'})$$

$$(((\exists\, i,\ B[i] = \text{`x}) \wedge (B[j'] = x) \wedge (present' = \textbf{true})) \bigvee ((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \neq x)) \wedge (present' = \textbf{false})))$$

$$(((\exists\, i,\ B[i] = \text{`x}) \bigwedge (B[j'] = x) \wedge (present' = \textbf{true}))$$

$$((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \neq x)) \bigwedge (present' = \textbf{false})))$$

$$(\forall\, i,\ ((1 \le i \le N) \Longrightarrow B[i] \neq x))$$

It is not more formal or more difficult than the programming language you use.

How can Documents be Both Precise and Readable?

This is readable

Set i to indicate the place in the array B where x can be found and set present to be **true**. Otherwise set present to be **false**

but vague and unclear:

- What do you do if the array is of zero length?
- What do you do if x is present more than once?
- Are you allowed to change B or x?
- What does the “otherwise” mean: Does it mean, if you don’t do what you should, or if there is no place in the array where x can be found, or if there are many places where x can be found?

We have all seen worse examples of such sentences.

How can Documents be both Precise and Readable?

This is readable and better than the text above:

Specification for a search program

	x can be found in B	x can not be found in B
j =	place where x can be found in B	any number at all
present =	**true**	**false**

How can Documents be both Precise and Readable?

This is precise and readable (by trained people).

Specification for a search program

	$(\exists\ i, B[i] = x)$ [a]	$(\forall\ i, \neg(B[i] = x))$
j' |	B[j'] = x	*true*
present'=	true	false

$\wedge$ NC(x, B)

a. These tables depend on using a logic that is defined for partial functions in which evaluating a partial function outside of its domain yields "***false***" for the set of built-in predicates.

1 The first can be input to mathematics based tools but is difficult for people.
2 The second seems clear but does not answer key questions.
3 The third is clearer but does not answer one key question and cannot be input to reliable tools.
4 The fourth is complete and could be processed by tools. It is, in theory, equivalent to the first, but in practice much better.

Tabular Notation is not Limited to Requirements Documents

We first “discovered it” in the requirements area, but it can be used for

- **module interface specifications**
- **module internal design documents**

These documents can be tested for completeness.
They can be used to generate prototypes.
They can be used to generate test oracles.
They make the programming *much faster* and *more reliable*.
But, most important:

They can be read by human beings.

They reduce the number of errors made by those who use them.

Dividing the Software to Conquer Complexity

Small modules are easier to understand, if the interfaces to other modules are simple.

To keep interfaces simple, “hide” the details inside the module.

Use the requirements documents to help structure the software:

- Some modules hide the requirements (REQ)
- some modules hide software decisions (which are not in the requirements document).
- Some modules hide the hardware (IN, OUT) These modules are support software.

These modules “create” virtual:

- data structures,
- devices,
- “actors”,

“objects” that do part of the job.

It is at this stage that we have the best chances for re-use - but we must document the interfaces.

Documenting Module/Object Interfaces (1)

It is wise to design software by designing a set of objects.

- An object is a finite state machine.
- Each object is implemented by a module (a set of programs) using a data structure that is “hidden from” (never used directly by) programs outside the module.
- Changing the state of the object, or getting information about the object’s state, is only done by invocations of programs from the module.
- The input alphabet of an object is the set of operations one can perform upon an object.
- The output alphabet of the object is the set of values that can be returned by such operations.

The representation of the state of an object can be hidden.

Describing or specifying objects is very different from describing or specifying programs.

Hiding the state means that we must discuss event sequences, but it makes future changes easier.

Documenting Module/Object Interfaces (2)

Black-box interface descriptions must be written in terms of (input, output) sequences (traces).

- A *trace* of a finite state machine is a finite sequence of pairs, each containing a member of the input alphabet and a member of the output alphabet.
- A trace, T, is considered *possible* for machine M, if M could react to the sequence of inputs in T by emitting the sequence of outputs in T.

Descriptions and specifications of objects can both be written as predicates on classes of traces.

These predicates can be described by an extension function/relation.

- For every finite trace and input, we define what the possible output is.

We organise our descriptions in terms of:

- A canonical[1] *abstract* state representation, and
- single event extensions of those traces.

Result: a systematic, reviewable reference document

[1] A canonical form is a normal form with the property that no two canonical expressions are equivalent.

Interface Documentation: 12 Element Queue: Syntax

ACCESS PROGRAMS

Program Name	Value	Arg#1
ADD		<integer>
REMOVE		
FRONT	<integer>	

Canonical representation

$$\text{rep} = < [a_i]_{i=1}^{n} > \wedge \ (0 \leq n \leq 12)$$

Interface Documentation: 12 Element Queue:

Trace Extension Functions[1]

ADD([rep],a) ≡

conditions	new rep	extension class
n = 12	rep	%full%
n < 12	rep.a	

REMOVE([rep]) ≡

conditions	new rep	extension class
rep = _	‘rep	%empty%
rep ≠ _	$<[a_i]_{i=2}^{n}>$	

FRONT([rep]) ≡

conditions	new rep	extension class	Value re-turned
rep = _	rep	%empty%	
rep ≠ _	rep		a_1

[1] We use “.” to denote sequence concatenation. [brackets] enclose implicit arguments to functions.

Design Reviews for Module Interfaces

Lots can be wrong with an “innocent” looking interface:

- The implicit assumptions can be wrong.
- The implicit assumption can be inconsistent.
- Interfaces can force inefficiencies on the system
- Interface assumptions can be likely to change
- Interface descriptions can be ambiguous

Interface decisions are early decisions.

Interface decisions affect more than one module.

Interface documents deserve serious thought!

They tend to be casually reviewed - a grave mistake because each change in an interface affects at least two modules.

The interface shown cannot efficiently reinitialize, remove the most recently introduced element, etc.

Subsequent examples include Q12INIT for initialisation.

Documenting Internal Design

If we already have the module specification, we need to document:

1. The complete data structure.
   - Often data elements are introduced piecemeal - we need know it all.
   - Brooks, “Show me your algorithms and I will ask to see your data structure; show me your data structure and I may not need to see your algorithm.”
2. The interpretation of that data structure (known as an abstraction function).
   - Every possible data state corresponds to a trace, but which one.
   - The abstraction function maps from concrete states to the abstract representation.
3. The effect of each program on the data structure.
   - This is done by H.D. Mills’ “program function” s mapping from concrete states to concrete states.
   - Such functions are often best described by tabular notation.

Queue: Internal Design (part I)

Data Structure Description and abbreviations.

CONSTANTS

Constant Name	Definition
QSIZE	12

TYPES

Type Name	Definition
<qds>	array[0..QSIZE-1] of integer

VARIABLES

Type Definition/Name	Variables	Initial Values
<qds>	DATA	“Don’t Care”
0..QSIZE-1	F, R	“Don’t Care”
<boolean>	FULL	“Don’t Care”
<boolean>	old	false

Abbreviations:

edge ≙ $(R = F + 1) \lor (F = QSIZE-1) \land (R = 0)$ ≙

<qs> ≙ $qds \times 0..QSIZE-1 \times 0..QSIZE-1 \times boolean$

The Abstraction Function[1]

af: <qs> → <queue12>

af(DATA,F,R,FULL,old) ≙

(¬ *edge* ∨ FULL) ∧ (F ≥ R) ∧ old	Q12INIT.(DATA[F]).(DATA[F−1]). … (DATA[R])
(¬ *edge* ∨ FULL) ∧ (F < R) ∧ old	Q12INIT.(DATA[F]). … .(DATA[0]).(DATA[QSIZE-1]). … .(DATA[R])
edge ∧ ¬ FULL ∧ old	Q12INIT
¬ old	

[1] The above table explains how the data are intended to be interpreted. It is redundant and used for checking.

Describing a Module's Programs

A program is a part of a module.
We wish to describe its effect on the module's private data structure.

We distinguish 3 types of descriptions:

- *constructive descriptions*, which show how a product is constructed from other products,
- *behavioural descriptions*, which describe the visible behaviour of a product without discussing how it was constructed, and
- *specifications*, which describe the requirements that a product must meet.

In my view this is a very important distinction that is ignored by the "formal methods" community.

Relational Program Descriptions and Specifications

Users need to know the relation between the starting values of variables and the final values of variables.

Users need to know the starting states for which the program is guaranteed to terminate.

We base our work on Harlan Mills' ("Cleanroom") program function, but

- Represent the function in a more readable tabular format.
- Deal properly with non-determinism.
- Carefully distinguish between relations as specifications and relations as descriptions.

It is possible to produce short, readable specifications of programs and review them before writing the actual code.

This forces designers to think about issues that they tend to overlook (such as error response).

Queue: Program Functions

pf_Q12INIT ⊲

F' =	0
R' =	1
FULL' =	false
DATA' |	true
old =	true

gpf_ADD(a) ⊲ NC(F) ∧ ∀j (j ≠ R') [NC(DATA[j])] ∧ NC(a) ∧

	(‘R = 0) ∧ old ∧			(‘R ≠ 0) ∧ old ∧			¬ old
	‘*edge* ∧		¬ ‘*edge*	‘*edge* ∧		¬ ‘*edge*	
	‘FULL	¬ ‘FULL		‘FULL	¬ ‘FULL		
DATA'[R'] =	‘DATA[‘R]	a	a	‘DATA[‘R]	a	a	‘DATA[‘R]
R' =	‘R	QSIZE-1	QSIZE-1	‘R	‘R – 1	‘R – 1	‘R
FULL' =	‘FULL	false	‘F = QSIZE-2	‘FULL	false	*edge*’	‘FULL

pf_REMOVE ⊲ NC(DATA,R) ∧

	(¬ ‘*edge* ∨ ‘FULL) ∧ old ∧		(‘*edge* ∧ ¬‘FULL) ∨¬ old
	(‘F = 0)	(‘F > 0)	
F' =	QSIZE-1	‘F – 1	‘F
FULL' =	false	false	‘FULL

pf_FRONT ⊲ NC(R,FULL, DATA, F) ∧

	¬ ‘*edge* ∨ ‘FULL old ∧	(‘*edge* ∧ ¬‘FULL) ∨ ¬ old
return value =	‘DATA[‘F]	

What Good are these Tables?

These tables are detailed designs for the programs.
They are more easily checked than code.
Table writers make fewer errors than direct coders.
Writing the code once you have the table is usually quick and reliable.
The tables make the language independent design decisions.
The tables are easy reference material.
The tables can be checked for correctness (commutative diagram)
The second table also shows how to simplify the code.
Test oracles can be generated.
Other testing functions are supported.
These tables support a systematic inspection process.
They take effort but that effort is far from wasted.

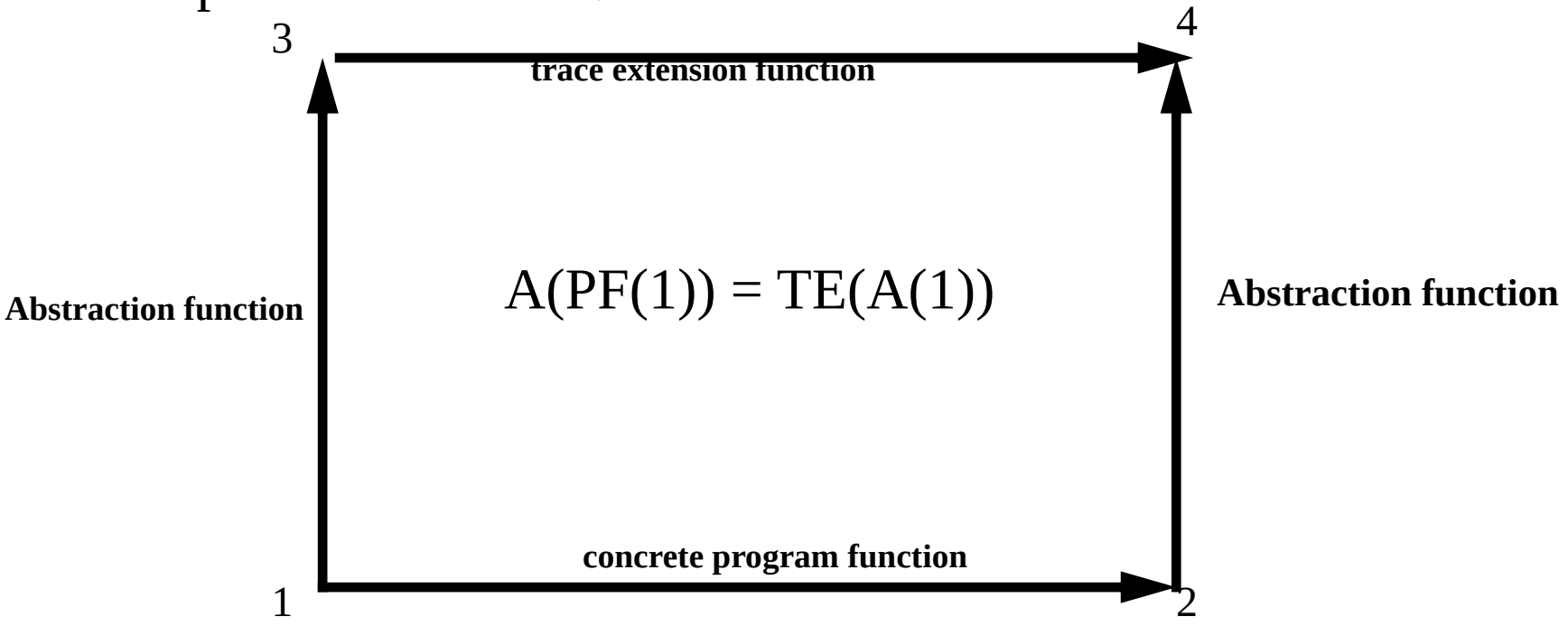
Checking A Module Design

A module design may be wrong, i.e programs that implement it won't work.

It is good to be able to check this before you invest in coding.

The module design can be checked against the module specification.

There are two paths from 1 to 4.



If the equation holds, the design can be made to work.

Imperfection of Documents?

When engineers work with physical products they must use imperfect implementations of abstract specifications.

With software, imperfection is not always necessary but it may be convenient and acceptable.

The imperfections must be “bounded” and explicitly limited in their applicability.

For example, we may ignore the limits on representations of numbers because we only work with a limited range of numbers.

It is important to include this in the specification.

No new mathematics, no special notation, is needed for this.

The use of mathematics in engineering does not imply a belief in perfection of programs or maths.

This is a red herring thrown out by those who do not want to succeed.

Displays

A display consists of 3 sections:

- A small program identified in the hierarchical decomposition
- A specification stating what that program must do.
- Specifications for the programs used by that program.

The program is kept small by removing code sections, creating a display for them, and including their specification in the bottom part. This is the hierarchical decomposition.

To check a display, determine the description of the program and see if it satisfies the specification. In doing this, use the specifications of the invoked programs, **not** their text.

To check completeness of the set of displays, check that every specification at the bottom of a display is at the top of another. The exceptions:

- standard programs - defined in standards documents.
- primitive programs - defined in company dictionaries

Finally, check the correctness of each display.

Completeness can be checked mechanically,

How Mathematics can help “Real” Developers

The value of a sound model

- Although the ideas I have just talked about arose in practical projects, finding a supporting “theory” has improved them and made tools possible.
- All of the practical projects could be done better now.
- People are “forced” to follow strict discipline and do a better job.

Documentation tools

The mathematics provides a precise meaning to these documents, which means that it is but a short step to tools that

- make it easier to produce the documentation
- check the documentation
- make the documentation more useful

Preliminary Applications: The Taste of Success

We do not have “production quality” tools, but

- Ontario Power Generation and AECL have industrial quality tools for special cases.
- U.S. Naval Research Laboratory has very powerful tools for the requirements application of the ideas.
- McMaster’s tools have demonstrated how much can be done by “hit and run” programmers (graduate students).
- Every application has led to improvements in quality.

Alternate Approaches

B, Z, VDM do not have the readability and, because they are not based on standard mathematics, do not have standard definitions.

UML does not have defined semantics, is being redefined, and growing.

B,Z, VDM, were theory first, look for usability afterward.

UML is a committee product.

Predicate wp transformer can be derived from our tables, but the reverse is not generally true.

None of the others have tabular notation.

Can we Test documentation?

Nobody trusts today's documentation.

- It is vague.
- It is incomplete.
- It is inconsistent.
- It is inaccurate.
- It is hard to find all the information you need.
- It is hard to know if you found everything.

If nobody trusts it, few value it.

If few value it, nobody invests effort in it.

If nobody invests effort in it, its no good.

If its no good, nobody trusts documentation.

Can we improve documentation by testing?

- Yes, if we can run tests that compare it with the software it documents, which we can do but...
- only if we have mathematical precision.

How to Interpret These Descriptions and Specifications.

Consider the following “Gedankenexperiment”.

- Save all the values of the state variables that are mentioned in a table in “before variable”.
- Run the program.
- Save all the values of the state variables that are mentioned in a table in “after variable”.
- Evaluate the boolean expression, that evaluates the predicate.

If the predicate evaluates to ***true***, the behaviour just seen was acceptable. Otherwise it is not.

Variables not mentioned are “don’t care” variables.

The “Gedankenexperiment” was actually a test.

The program that evaluates the predicate is a test oracle.

The program that produces the oracle from the tabular documentation is a test oracle generator (TOG).

Traces as Test Cases

It is popular to replace documents with use-cases or scenarios.

A trace specification gives a complete set of use-cases.

This is also a set of test-cases.

This too is the basis for an oracle generator.

Can “Real People” Learn This Stuff?

Many academic ideas can only be used by Ph.D. students of the inventors.

Requirements ideas have been used many times.

Basic program-function concepts introduced by Harlan Mills at IBM with measurable success when first introduced - even without the tabular notation.

Ontario Hydro uses these ideas for safety-critical applications.

Students in McMaster’s new Software Engineering programme use it with great success in their projects in second and fourth years.

U.S. Naval Research Laboratory has introduced ideas in military projects.

Even without the mathematics, we get great improvement.

First year engineering students have learned to read and implement from specs.

Tabular notation - no theoretical advantage, but a great practical advantage.

People can apply the inspection technique after a 3 - 4 day course.

Critical Mass in a company is essential. Writers without readers are useless.

There is lots of room for improvement. We will identify these faster if you work with us.

The Next Steps

Tools to make the production of these tables easier,

- input tools
- formatting tools
- completeness and consistency testing tools

Tools to manage inspections

- Keep track of the displays that have been checked
- Keeping the

Testing tools

- test oracle generators, monitor generators
- test case generators,
- reliability prediction tools
- test coverage evaluation tools

Verification tools.

- composition, union and functional-equality tools.

Industrial case studies to test and improve the tools.