Software Documentation: The Real Computer Science Grand Challenge

David Lorge Parnas

Abstract

For many years it has been fashionable to propose rather grandiose applications as "grand challenges" for computer science. This talk discusses a more "down-to-earth" problem, one that must be solved before we will be able to make real progress in the area of software quality.

To produce software that we can trust, we must be able to provide precise documentation that can be used as a trustworthy source of necessary information about the structure of the system and the behaviour of each of its components. We discus why this is needed and how research can contribute to ameliorating this problem.

An Old Old Problem

Apeldoorn - 1969: My manager told me that they were unable to write specifications for software components that were sufficiently complete and precise that components would work properly together when completed.

Other 1969 problems:

- **Unable to state what a program must do other than by writing one.**
- **Unable to fully document visible behaviour of a computer system.**
- **Unable to identify components affected when change is needed.**
- **Unable to describe a component's "design" other than in code.**
- **No basis for preparing tests in advance.**
- **Hard to inspect complex products.**

These problems considered urgent, solution essential!

A Very Current Problem

None of the old problems have been solved and, ...

Microsoft is being fined millions per day because they have not been able to document the interfaces to their servers.

- **The EC wants a document sufficient to allow a competitor to build a compatible (interoperable) competitive server.**
- **They claim that source code is the most precise and authoritative documentation possible (see http://www.microsoft.com/presspass/press/2006jan06/01-25EUSourceCodePR.mspx)**
- **At best code tells you what it does, not what it should do.**
- **At worst, you can't read it accurately.**

It seems obvious (but hard to measure) that our inability to provide good professional reference documentation is costing us millions maintaining complex products.

The Role of Documents in Engineering

Engineers Design Through Documentation

Documents record key design decisions

- to guide the builders
- to assist in inspection
- to assist in maintenance
- to enable review

Documents are binding on everyone and fully controlled.

Documents are precise documents that use mathematics.

Documents are not introductions or tutorials,

Documents are not extracted comments (javadoc).

Documents show "separation of concerns".

The 40 Year "Software Crisis"

Obviously not a "crisis".(turning point, moment of truth)

Three underlying causes for well-known symptoms:

- **Lack of disciplined design, careful choice of interfaces and careful structural decisions**
- **Lack of thorough, review, inspection and testing**
- **Lack of accurate, complete, precise documentation**

Other problems (bugs, security flaws, cost of change, poor team cooperation, • • •) are all secondary.

We must attack the cause, not just the symptoms.

Documentation is the key to all three.

Why Documentation is the Key to Solving These Problems

- **Decisions that are not fully documented have not really been considered and taken.**
- **Decisions that are not documented cannot be reviewed and products cannot be tested for compliance.**
- **Hierarchical Documentation is the key to inspection.**
- **Precise documentation can be used to generate test-cases and evaluate test-results.**

The best thing that we researchers could do for software developers is to develop good documentation methods and tools to support the use of those methods and documents.

What Do We Mean by "Documentation"

Practical tool, not just a theoretical achievement.

- **Authoritative repository of information**
- **Usable as a quick and reliable reference by developers, reviewers, maintainers, users.**
- **Easier to use for information retrieval than the code**
- **Structured to avoid inconsistency**
- **Quicker and more authoritative than trial executions**
- **Useful before, during, and after the coding.**

Computer Science and Software Documentation

Engineers, managers bemoaned inability to document (specify and describe the behaviour of) software.

Well known people responded by publishing specifications that were fragments of programs - not statements of requirements for that program.

Computer Scientists have treated specification as meaning "programming in a useless language" or models since that time.

They regard documentation as an essay or commentary.

If you take the word "Engineer" in Software Engineer" seriously, you must ignore such views.

Five Requirements for Software Documents

- Accuracy
- Precision
- Consistency
- Completeness
- Ease of reference.

The first three can be achieved by using mathematics.

The last three require improved notation and organization.

All require content definitions for each document.

Some Research Questions:

What documents are needed and when are they needed?

What contents are required for each document?

How can we best represent this information (notation)?.

How can we check documents for consistency and completeness?

How can we show the workability of designs?

How can testing compare software and documents?

What tools would help to produce and use documentation

How can we specify requirements for interoperable network servers?

Basic Documentation Guidelines

Never mix reference documents with introductions.

Never rely on words; they will never be precise enough.

Mathematics is the only way to be precise, but

- expressions must be simple and easily parsed
- Interpretation should be direct (closed form).

Only relevant information should be included and this information should be stated as directly as possible.

Each item of information should be in only one place and everyone should know where it will be put/found .

These are all easier said than done.

Document Roles

Engineering usage:

- A *description* states properties of a product; it may include a mixture of incidental and required properties.
- A *specification* is a description that states only required properties.
- A *full specification* is a specification that states all required properties.

The same notation may be used for all 3.

These classifications are a matter of intent not notation.

There is no such thing as a "specification language".

Is Mathematical Documentation just "Formal Methods"?

Our goal is to organize information for easy retrieval.

Engineering style of mathematics ("closed-form" solutions)

Not axiomatic in style. Users calculate, not derive.

No new mathematics needed - classical concepts suffice.

Not intended for automatic programming (but possible).

Proof is not the main goal (but possible).

Documents are neither models nor programs.

Content Definitions: Why and How

Organizations often specify the format of a document, but they rarely specify the content.

We need a content definition to

- know what to put where
- check for completeness.

Each document describes some mathematical relations.

Each document has a different range and domain

Define document content by stating which relation.

A complete document specifies which pairs are contained.

Content definitions say NOTHING about format, structure, notation, etc.

The Main Documents

(1) "System Requirements Document": Treats computer system as "black-box".

(2) "System Design Document": Describes computers and communication.

(3) "Software Requirements Document" [(1) + (2) or (1) - (2)].

(4) "Software Function Specification": Describes actual software behavior.

(5) "Software Module Guide": How to find your module. (informal document).

(6) "Module Interface Specifications": Treat each module as black-box.

(7) "Uses Relation Document": Range and domain are sets of programs.

(8) "Module Internal Design Documents": data structure, abstraction relation, and program functions.

Other documents may be needed for some systems.

- process structure
- component communication structure

Example of a Functional Definition of Document Content

Deterministic Program effect described by a function from data-states to data-states.

(x,y) in that function if a program started in state x will terminate in state y. Function may be partial.

- **Function can be described by precondition/postcondition pair.**
- **Function can be described by Dijkstra's "wp" (weakest precondition).**
- **Function can be described by predicates (Hehner).**
- **Function can be described by concurrent-assignment (Mills).**

More is needed for non-deterministic programs.

- **relation (data states ⇒ data states)**
- **competence set (starting states with termination guaranteed)**

Notational Conventions:

If R is a relation,

(1) R refers to the set of ordered pairs that constitutes the relation,

(2) R (x,y) is a predicate, the characteristic predicate of the set R. R(x,y) is *true* if and only if (x,y) is in R.

(3) If R is a function, R(x) refers to y such that R(x,y)

Documenting system requirements? (I)

Identify monitored variables ($m_1, m_2, \cdots, m_N$).

Identify controlled variables ($c_1, c_2, \cdots, c_P$).

For each scalar variable, x, denote the time-function describing its value by "x^t".

The value of x at time t is denoted "$x^t(t)$".

A vector of time-functions ($v^t_1, v^t_2, ..., v^t_n$) is denoted by "$\underline{v}^t$".

How Can We Document System Requirements (II)?

Describe the following relations:

Relation NAT:

- **domain contains values of m^t,**
- **range contains values of c^t,**
- **(m^t, c^t) is in NAT if and only if nature permits that behaviour.**

Relation REQ:

- **domain contains values of m^t,**
- **range contains values of c^t,**
- **(m^t, c^t) is in REQ if and only if system should permit that behaviour.**

How can we document system requirements? (III)

The specification is complete if:,

- **(1) domain(REQ) ⊇ domain(NAT).**

The relation REQ can be considered *feasible with respect to NAT* if (1) holds and:

- **(2) domain (REQ∩NAT) = (domain(REQ) ∩ domain(NAT)).**

How can we document system design?

i^t denotes the vector valued time function $(i^t_1, i^t_2, \bullet\bullet\bullet, i^t_r)$
one element for each of the input registers
o^t denotes the vector valued time function $(o^t_1, o^t_2, \bullet\bullet\bullet, o^t_q)$
one element for each of the output registers

Document the following relations

Relation IN
domain contains values of $\underline{m}^t$
range contains values of $\underline{i}^t$
$(\underline{m}^t, \underline{i}^t)$ is in IN if and only if input device permits that behaviour

It must be the case that
(1) domain(IN) ⊇ domain(NAT)

Relation OUT
domain contains the possible values of $\underline{o}^t$
range contains the possible values of $\underline{c}^t$
$(\underline{o}^t, \underline{c}^t)$ is in OUT if and only if output device permits that behaviour

How can we document software requirements?

Software requirements = system design + system requirements
(1) $REQ(\underline{m}^t, \underline{c}^t)$
(2) $IN(\underline{m}^t, \underline{i}^t)$
(3) $OUT(\underline{o}^t, \underline{c}^t)$ and
(4) $NAT(\underline{m}^t, \underline{c}^t)$

The actual software can be described by
(5) $SOF(\underline{i}^t, \underline{o}^t)$

For the software to be acceptable, SOF must satisfy:
(6) $\forall \underline{m}^t \forall \underline{i}^t \forall \underline{o}^t \forall \underline{c}^t [IN(\underline{m}^t, \underline{i}^t) \wedge SOF(\underline{i}^t, \underline{o}^t) \wedge OUT(\underline{o}^t, \underline{c}^t) \wedge NAT(\underline{m}^t, \underline{c}^t) \rightarrow REQ(\underline{m}^t, \underline{c}^t)]$

Using functional notation:
(6a) $\forall \underline{m}^t [\underline{m}^t \in domain(NAT) \rightarrow (REQ(\underline{m}^t) = OUT(SOF(IN(\underline{m}^t))))]$

Modules and Components

Distinct but related concepts:

Module (historically):

- **tasks (work-assignments) for developers**
- **should have a shared "secret", (design decision) that it hides**

Component:

- **Distribution unit - use as a whole or do not use it**
- **should offer an integrated set of services**

A module may include several components

A component may include (parts of) several modules

Often something is both a module and a component.

Same documentation scheme for both.

Module/Component Interface Documents

Event descriptors: (Description of pre/post values of visible variables)

Time is included as a variable if needed.

Trace: Sequence of event descriptors

History: Trace describing what actually happened

Document Relation: (history ⇒ output values)

Closer to our (A-7, SCR, 4-variable) requirements method than to earlier trace methods

How can we document internal module design?

Module = private data structure + set of externally invocable programs

Design, usually in designer's head, should be written down

Three essential elements:

(1) description of the data structure

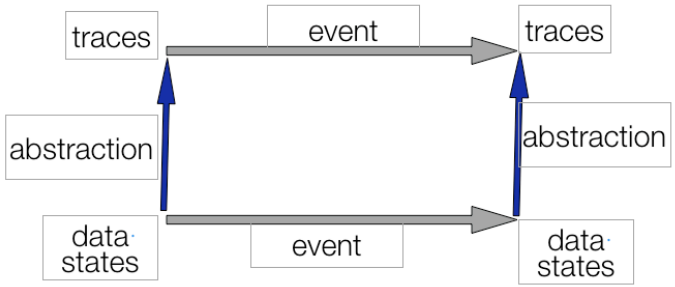
(2) abstraction relation (AR)

- **Domain: states**
- **Range: traces**
- **(d, t) is in AR if history t could result in state d**

(3) program function (LD-relation) for each externally invocable program.

How can the workability of a design be verified?

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



AR(d1,t1) ∧ e(d1,d2) = AR(d2,t1.e)

Notational Issues: How to Describe Relations

Relations can be described by a characteristic predicate but

- **What will it look like?**
- **Will people be able to find what they want?**
- **Will we make lots of errors?**
- **Will people read it correctly?**

The answers to these questions will discriminate between theoretical achievements and practical ones.

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))

Keyboard Checker: Tabular Expression

N(T) =

keyOK			T=_	¬ (T= _) ∧ N(p(T))=1	¬ (T= _) ∧ 1<N(p(T))< L	¬ (T= _) ∧ N(p(T))= L
				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))

Two Distinct Ideas

1.
2. Tabular Expressions
3. Relational Model of Documentation

You need both ideas!

- **The relational model eliminates irrelevant "models" and focusses the documents on observable behaviour.**
- **The tabular expressions "parse" the expressions and make it possible to use the information easily.**

Practical Experience

Major Projects:

- **A-7 Onboard Flight Program**
- **Darlington Nuclear Station Shutdown System (60 + people)**
- **Bell Laboratories Service Evaluation System**
- **Ericsson Base Station Component**
- **Dell Keyboard Checking Program**

A wide variety of smaller projects and examples

Conclusions:

- **It requires training!**
- **Engineers can use it and like it - especially readers.**
- **Fewer mistakes than other approaches**
- **Many practical benefits**

Research Problems Revisited

- Additional documents with definitions
- Still better notation
- A set of practical tools (I failed!)
- Table checking algorithms
- Intra-document consistency checking
- Test case generation
- Test oracle generation
- Inspection management systems.
- Moving from scenarios/traces/use cases to complete documents
- Dealing with concurrency
- Standards for documentation, testing, inspection
- Document-based cooperative work systems
- Using documents in verification and inspection • • • • •

Software Quality Research Laboratory - University of Limerick - Ireland

Summary

In Software Engineering we can and should take documentation as seriously as they do in other Engineering Disciplines.

We can define the contents of documents

We can check the feasibility of a design before implementing.

We can use traditional mathematics in traditional ways

No new “specification language” is needed, but we can improve the notation used for mathematical expressions.

There is a world of useful research problems waiting for younger people.