Communication
Explaining and Documenting Software

Three Types of Communication:

Oral Communication

- Usually face-to-face with discussion possibilities.
- Presenter can observe the audience and adjust to their reaction.
- Presenter can accommodate audience needs/wishes.

Written Text:

- Intended to stand on its own.
- Not interactive in any way.
- Cannot adjust to audience needs
- Writer must anticipate audience.

Visual Aids (Overheads, slides, ...)

- Intended to supplement oral presentation
- need not stand on its own
- helps the audience to follow your structure
- repeats the most important points
- can provide illustration
- allows you to go too fast.

Types of Communication

Reference Documentation for Maintainers

Introductory Documentation for Maintainers

Reference User Documentation

Tutorial User Documentation

Overview User Documentation.

Comparative Reports

Investigatory Reports

Publicity Communication

Advertising

The Most Basic Rule for All
Know Your Audience

- Write down what you expect them to know.
- Write down what terms they know.
- Write down their purpose in listening or reading.
- Write down your purpose in communication.
- Write down what you want them to know afterwards or what you want them to be able to do.

You must spend time on this.

- Find out where they learned what you expect them to know.
- Make sure that they do understand the vocabulary
- Get agreement from your customer.

Use this “requirements document” when you prepare your communication.

The Second Rule:

Design your communication's structure before you prepare the words.

Formulate the questions you are planning to answer.

Refine your "question outline" top down.

Show that answering all low level questions will answer the question on the next level up.

When preparing make sure that your audience can follow you. If you are assuming that they know something it must either be a prerequisite or it must be something you have already told them.

This is a program design problem. You must know the starting state, the desired end state, and make sure that you have a sequence of state changes that will execute in the processor - your audience.

Documentation Guidelines

Why design documentation is important.

Uses during development:

- Communication among designers, users, programmers, etc.
- Training - makes personnel turnover less disruptive.
- Prevents duplication of effort - if reasons for design decisions recorded, reduces need to rethink them later.
- Basis for design reviews.
- Quality assurance - standard against which software can be judged.

Uses during maintenance:

- Training
- Reduces labour of evaluating feasibility of changes.
- Guides programmers as they find and correct errors.
- Repository of design information, which even the original programmers often forget.
- Preservation of program conceptual integrity - maintenance programmers have a way to check consistency of proposed changes.The existence of alternatives when something goes wrong

Common Problems with Documentation Why is it Hard to Use?

Difficult to understand - assumes reader knows more than he/she does.

Difficult to find answers to specific questions.

Difficult to maintain - gets out-of-date all too soon.

Wordy, repetitive, and boring.

- Confusing, inconsistent terminology
- People appreciate clear, concise termination.

Remedy

View documentation as the most important product of design, not as a by-product of coding.

Design the documentation - objectives, contents, organisation, format.

- To be a convenient format for designers to record and exchange ideas.
- To serve as ready reference tools.
- To be maintained - controlled and kept up-to-date.
- To explain reasons for decisions since reasons cannot be inferred from code.

General Principles for Documentation Design

Determine objectives

- Who will need it?
- What should they already know?
- What should they be able to find out?

State questions before trying to answer them.

Separate concerns.

Documentation should consist of mutually supportive formal and informal parts.

- Formal - precise, concise, unambiguous
- Informal - provides a guide to the formal

General Principles for Documentation Design

Warning:

If you are not careful, people will depend on the informal documentation and ignore the precise documentation.

This leads to misunderstandings.

Keep the two complementary.

- Use English only for overviews, narratives, and explanations.
- Use abstract Programs (otherwise know as PDL or coding specifications) for documenting algorithms.
- Use mathematics to describe functions and relations.

When you are presenting lots of information:

Use the questionnaire method: Design forms, tables, notation, templates.

Careful design of forms, tables, and similar formal document structures will assure:

- complete coverage rather than haphazard coverage
- a well structured logical organisation for the information
- consistency in the information that is presented and the way that it is presented.
- Areas of incompleteness that are known.
- Ease of review.

When you discover that the form is not right:

- correct the form
- review all earlier work.

Do not make *ad hoc* variations from the forms.

Do Not Confuse the Following Types of Documentation

1. Software Requirements Specification (e.g., Programs Performance Specification)
2. Overall Design Documentation
3. Module Interface Documentation
4. Module Internal Design Documentation
5. Program Design Documentation

Writing Down Requirements

The most costly errors are those made early in the process - they are the hardest to change.

Misunderstandings about requirements lead to early mistakes. Those are costly mistakes.

Programmers need to be told what is needed.

They must also be told what is subject to change.

Requirements must be subject to review.

Safety reviews of software must be based on a previously agreed statement of requirements.

Maintenance actions must be based on requirements.

None of these things is possible unless we have a *written* statement to work with.

That *written* statement must be precise and complete.

The first responsibility of the “Software Engineer” is to obtain an accurate and complete statement of requirements.

How to document system requirements

The first step is to:

Identify monitored variables (m_1, m_2, •••, m_n).

Identify controlled variables (c_1, c_2, •••, c_p).

The primary monitored variables are things outside the system whose values should influence the output of the system. Examples:

- customer meter reading
- steam temperature
- time of day

The primary controlled variables are things outside the system whose values should be constrained or controlled by the system. Examples:

- what the operator sees
- what appears on a bill
- the temperature of the water.

This is only the beginning, but for many projects you cannot even find a complete list of these variables and there is no agreement on what they are.

Monitored and Controlled Variables Will Be Added During The Design Process.

It is inevitable that the need for additional variables will be discovered as we get into detailed work.

Further, new monitored and control variables are created during the design process.

The primary monitored and controlled variables are outside the system. Secondary variables may be internal.

- Sometimes we want to monitor the system itself, i.e. measure things that did not exist before the system was built.
- Sometimes we may even want to control (adjust) parts of the system.

As the design is developed, we may add these monitored and controlled variables to the requirements document.

It is essential that the document be updated as design continues. Not keeping documents up to date costs you more than it saves.

A Mathematical View of Requirements

The implementors need to know the following relations:

Relation NAT:

- domain contains values of $\underline{m}^t$, range contains values of $\underline{c}^t$,
- ($\underline{m}^t$, $\underline{c}^t$) 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 $\underline{m}^t$, range contains values of $\underline{c}^t$,
- ($\underline{m}^t$, $\underline{c}^t$) 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.

If we can describe these relations, we have our system requirements written down.

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

How can we document system design?

$\underline{i}^t$ denotes the vector valued time function
$(i^t_1, i^t_2, \bullet\bullet\bullet, i^t_r)$ with one element for each of the input registers

$\underline{o}^t$ denotes the vector valued time function
$(o^t_1, o^t_2, \bullet\bullet\bullet, o^t_q)$ with 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
domain(IN) $\supseteq$ domain(NAT)

Relation OUT

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

When Can We Skip System Design?

Sometimes the I/O devices are simple and we can have simple relationships between the controlled and output variables as well as between the monitored and controlled variables.

In that case, we can use the systems requirements document as a software requirements document.

Many applications have this property.

In some, we can cheat and mix the two.

Documenting Module/Object Interfaces (1)

It is wise to design software as a set of objects.

- 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.
- Every object is a finite state machine.
- 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 state representation of objects should 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.

Interface Documentation: 12 Element Queue

(1) Syntax

ACCESS PROGRAMS

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

(2) Canonical representation

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

(3) 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.

Documenting Internal Design

We need to document:

- The complete data structure.
- The interpretation of that data structure (known as an abstraction function).
- The effect of each program (program function or LD-relation)

Queue12: Implementation 1 - Pascal

(1) DATA STRUCTURE

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"

Lexicon:

$edge \stackrel{df}{=} (R = F + 1) \vee (F = QSIZE\text{-}1) \wedge (R = 0)$
<qs> $\stackrel{df}{=}$ qds $\times$ 0..QSIZE-1 $\times$ 0..QSIZE-1 $\times$ boolean

(2) ABSTRACTION FUNCTION

af: <qs> $\rightarrow$ <queue12>

af(DATA,F,R,FULL) $\stackrel{df}{=}$

$(\neg\, edge \vee \text{FULL}) \wedge (F \geq R)$	(DATA[F]) (DATA[F–1]). …. (DATA[R])
$(\neg\, edge \vee \text{FULL}) \wedge (F < R)$	(DATA[F]). … (DATA[0]) (DATA[QSIZE-1]). …. (DATA[R])
$edge \wedge \neg\, \text{FULL}$	<>

Access program functions will be found on page 23

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 using 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).

Internal Design (continued)

(3) PROGRAM FUNCTIONS

pf_Name	Arg#1	Value
pf_Q12INIT		<qs> → <qs>
gpf_ADD	<integer>	<qs> × <integer> → <qs>
pf_REMOVE		<qs> → <qs>
pf_FRONT		<qs> → <qs> × <integer>

gpf_ADD(a) $\stackrel{df}{\equiv}$ NC(F) ∧ ∀j (j ≠ R') [NC(DATA[j])] ∧ NC(a) ∧

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

pf_REMOVE $\stackrel{df}{\equiv}$ NC(DATA,R) ∧

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

pf_FRONT $\stackrel{df}{\equiv}$ NC(R,FULL, DATA, F) ∧

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

Imperfection in Documents

One excuse for not preparing such documents is that we cannot get them right.

When engineers work with physical products they must use imperfect implementations of abstract specifications. Exactness is often impossible.

With software, imperfection is not impossible 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 is needed for this. Implication does the job.

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

A valuable Special Case: Systems Characterised by Modes and Current Values.

For many systems, only a little of the past history is relevant.

This can often be summarised by identifying “modes of operation”.

There will often by a small finite number of mode classes each with a small finite number of mode states.

The current mode in each class can be defined by transition tables.

The controlled values are then a function of the current mode and the current inputs.

For this class of systems, we can build monitoring test systems.

We can use other summaries of the past history, such as average values, too.

Modes And Their Use

Understanding the modes can make a complex system seem simple.

Modes are classes of states:

- There are too many states to deal with directly.
- The actual states are implementation dependent.
- Modes characterise the history of the system.
- The purpose of modes is to simplify the function descriptions. Choose them accordingly.

There can be several classes of modes.

There can be interactions (excluded combinations). These should be minimised!

Mode transitions are caused by events.

Modes can be defined by transition tables

Mode Transition Table

	Airplane	ATV	Submarine
Airplane		@T(weight on wheels)	@T(wet)
ATV	@F(weight on wheels) ***when*** ¬ wet		@T(wet)
Submarine	@F(wet) ***when*** ¬ (weight on wheels)	@F(wet) ***when*** (weight on wheels)	

Assumption:

weight on wheels, and wet are detectable conditions.

Mode tables are often the best way to explain a confusing system to a user.

They are a divide and conquer technique

- Separate the mode transition rules from other behaviour.
- Deal with the modes one at a time.

Two Views of Modes

Modes are classes of event histories.

- Each Mode Class corresponds to a partitioning of the set of event histories.
- Each Mode in a mode class is one of the partitions of that partitioning.

Modes are classes of system states

- Each Mode Class corresponds to a partitioning of the set of system states.
- Each Mode in a mode class is one of the partitions of that partitioning.

In a well designed deterministic system these are equivalent black box and clear box views.

One of the most popular CASE tools “Statemate” supports this work.

Statemate’s semantics is too complex.

Displays

When you explain or document code, use the concept of a display.

The top part of each display is the specification for the program in the middle.

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

The bottom part contains a specification of these invoked programs.

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

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

- standard programs
- primitive programs

Displays can be formal or informal.

Completeness can be checked mechanically.

Essential Point: Divide and Conquer

The initial decomposition is essential. Attempts to simply scrutinise the program fail.

Trying to read the program the way a computer would is much less effective. Logically connected parts may be far apart.

The use of tables is essential. It breaks things down into simple cases so that

- We can be sure that all cases are covered
- Each case is straightforward

We consider all variables, but one at a time.

We consider all cases, one at a time.

We can take “breaks”, go home and sleep, even take holidays, without losing our place.

Using displays and tabular summaries is far more work than English paraphrasing, but it imposes a discipline that helps.

Using Information Theory to Improve Your Communication.

The information in a statement is related to its probability.

- Today I saw Dr. Taylor wearing a tie.
- Today I saw Dr. Parnas wearing a tie.

If everyone agrees with a statement it contains no useful information.

- Canadian Politician: “We will listen to Canadians.”
- “The class of finite state machines is large and varied”

Use the negation test:

- Consider various negations of a statement and ask if anyone would say them.
- If nobody would say the negation reconsider the statement.

Ask what you were really trying to say.

- The Liberals do not listen to Canadians.
- This thesis is about the design of finite state machines.

These translations do pass the negation test and clearly state what you want to state.

Take Documentation Seriously
Design Documentation Reviews and Configuration Control Procedures

Design reviews: What questions should reviewers ask themselves to determine if document meets its objectives?

Configuration control procedures:

- How are changes reported?
- Who decides whether to make them?
- Who reviews them?
- How are updates distributed? To whom?
- What tools are needed? - word processing support invaluable.
- Look at configuration management systems and versioning systems.
- Keep document versions and code versions aligned.

Process Documentation

In Engineering, especially with complex systems, processes must be defined and followed.

You won't follow it properly unless it is documented.

You may have to prove that it is documented.

You need precise milestone definitions - not just a few brief words.

- Under time pressure people take short-cuts.
- People often do not know what needs to be done.
- You must be able to prove that you are done.

Example: Y2K inspection

- inspect all programs
- check for key words
- make sure there are no dates being processed.

The above are all inadequate.

You can do them well or do them badly.

The best approach is to define work products.

- Define the required content of work products.
- Describe how work products will be tested or otherwise verified.

Using Abstraction

Distinguish between what is relevant to your reader/ listener and what is not.

Define abstractions that allow you to focus on what is relevant.

Explain the abstractions.

Finally, give the information.

Examples:

- Explain the modes of a device, then give the transition rules.
- Introduce a set of lists, then describe the content.
- Introduce the classes of objects, then define the effects of commands.