Advice on Documentation (Faking it)

Daniel M. Berry

Introduction -1

We are talking about documentation for the benefit of the designers, programmers, testers, and maintainers.

We are not talking about documentation for the user, who gets a user’s manual, which is an entirely different animal.

Introduction -2

Requirements specifications are documentation for the designers!

Introduction -3

This advice is derived from

**“A Rational Design Process:
How and Why to Fake It”**

**by David L. Parnas
and Paul Clements .**

Rationality -1

Programmers would like to think of themselves as rational.

Methodologists would like to believe that all programmers can be taught to be rational.

All would like to believe that rational programmers write good software!

Rationality -2

Methodologists write papers and books describing how to use their methods to write code rationally.

All of these papers and books have examples of nice, clear, step-by-step rational developments of code from requirements.

Rationality -3

Funny thing is that these authors probably revised their examples as much of the rest of us!

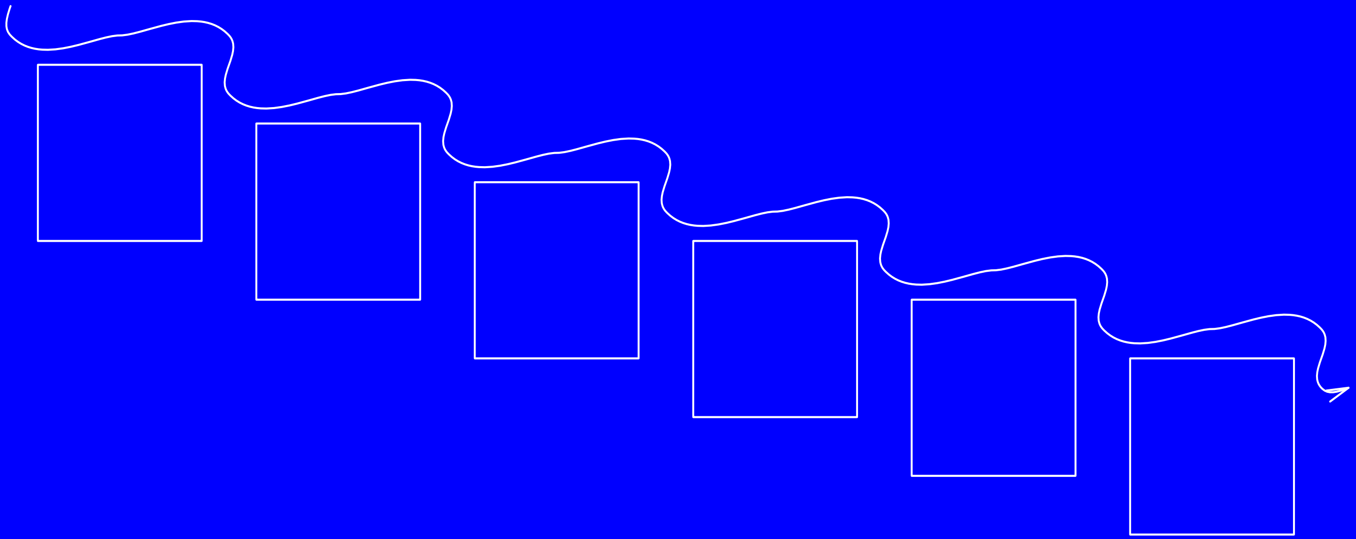
I know; I have written such a monograph.

I revised the requirements as much as I revised the development that supposedly followed them.

The same applies to lecturers on software engineering methods.

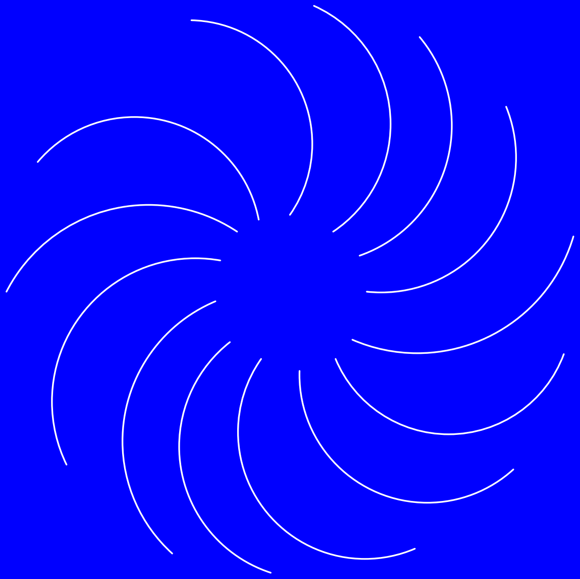
Rationality -4

Methodologists would have you believe that good programmers actually follow some variation of the waterfall lifecycle or some such.



Rationality -5

The reality is closer to the hurricane model.



Rationality -6

In both models you get wet, but a hurricane is much wetter and messier.

In addition, in the eye of the hurricane, there is a false sense of calm.

Rationality -7

These are the facts of life even in the most rational of all disciplines, mathematics.

No one discovers theorems as they are published.

No one discovers proofs in the way they are published.

There are lots of false starts, errors, scribbling, comments, proof-reading (in both senses), revision, etc.

Rationality -8

So the so-called rational software process is and will always be an idealization!

Why?

Clients do not really know what they want.

Even if all the requirements are known, many facts needed to complete development are discovered only during development.

Rationality -9

Even if all the facts are known, humans are incapable of fully comprehending all the details that will need to be taken into account.

Even all details are mastered, projects are subject to change.

Human errors can be avoided only if we do not use humans, but then who would program?

Rationality -10

We often have preconceived design ideas, and we invent ideas not obtained rationally.

We would not want to completely avoid the latter because this is where creativity happens!

For economic reasons we reuse or share code, and the reused or shared code is not ideal for the present use.

Rationality -11

OK, OK, so process is *not* rational. Nu?

But, there is value to describing the development of software as if it were rational, i.e., of faking a rational process.

"I know that I've been fakin' it!"
— Paul Simon

Faking It -1

Write the documentation as if the development were rational.

Be prepared to modify it, when the development changes direction as the developers get too wet in the storm.

Faking It -2

Given this, why bother taking courses like this one?

You cannot *fake* it unless you know the ideal that you are trying to fake.

Also, while being rational over the whole process is hopeless, being rational over small pieces of the process is possible and is helpful.

Faking It -3

Why is it useful to fake it?

Designers need guidance; a good understanding of the ideal process tells designers how to proceed, especially if they have designer’s block.

We will come closer to an ideal process if we attempt to follow it than if we proceed on an *ad hoc* basis.

Faking It -4

In any organization, it pays to have standard procedures; it is easier to move people, ideas, and code.

Once you have a standard, it might as well be ideal.

If there is an ideal process, it is easier to measure progress and to review products along the way.

Faking It -5

For each stage of a software development, the description of the ideal process should tell us

- **what to work on next,**
- **what criteria the work product should satisfy,**
- **what kind of persons should do the work,**
- **what information they should use in the work.**

Requirements Documents -1

We are talking about client-approved requirements documents.

Why do we need them?

Writing them makes us less likely to make requirements decisions accidentally.

They help avoid duplication and inconsistency in descriptions of what the software is to do.

Requirements Documents -2

They help teach programmers the application area.

They are necessary but not sufficient for making resource estimates.

They help insure against ill effects of personnel turn-over.

Requirements Documents -3

They provide a basis for an independent test plan development, especially the tests of achievement of functionality.

They provide constraints for future changes to system.

They can be used to settle arguments among the developers without having to go back to the client every time.

Requirements Documents -4

What should be in the requirements documents?

Basically, they should contain everything you need to know to write correct software and *no* more.

Every statement in them should be correct for all acceptable products; none should depend on any implementation decision.

Requirements Documents -5

They should be complete in that if a program satisfies the documents, the program is acceptable.

When information is just not known, the documents should say so rather than just leaving it out.

Requirements Documents -6

They are organized as reference documents rather than as introductory narratives about the system.

Reference documents are designed so that it is easy to look things up in them.

Requirements Documents -7

Requirements documents should cover the following topics, all specifications of properties of the software:

- **the machine on which the software runs**
- **the interfaces with the outside world, including users**

Requirements Documents -8

- for each output, its value as a function of the software-detectable system state
- for each output, how often or how fast it must be computed
- for each output, how accurately it must be computed

Requirements Documents -9

- **if the system is likely to change (and which system is not?), the areas most likely to change, to provide basis for trade-off decisions**
- **responses to undesired events that keep the software from fulfilling its complete requirements**

Documents for Faking It -1

The documents for faking a rational design process are

- **centered around modules**
- **supposedly obtained by following a rational method such as proposed by Parnas *et al*.**

Documents for Faking It -2

Module documentation:

- **module structure**
- **module uses hierarchy**
- **module interfaces**
- **module internal structure**

Documents for Faking It -3

Module structure:

includes exported procedures

Module uses hierarchy:

module A uses module B if and only if the correctness of A depends on the presence of a correct B in the system

Documents for Faking It -4

Module interfaces:

list of parameter
types
and meanings
and return types

for the module and all of its contained procedures

Documents for Faking It -5

Module internal structure:

> **all the design decisions encapsulated by the module, i.e., the *secrets* of the module**

Note that these are the things that were deemed likely to change.

So this list should include the list of possible changes that helped you design the modules.

NOTE: I do *not* want these secrets in the pre-goodie!!!

Pre-Goodie

In fact for the pre-goodie, I want only the externally visible stuff:

> **NO hidden data**
> **NO private data**
> **NO protected data**
> **NO secrets**

Only the stuff the user of the module needs to know.

Naming Types, Objects, and Procedures

The following suggestions are based on making the module descriptions as readable as possible, making as much of their semantics immediately apparent so that additional commentary is less important or even unnecessary.

Naming of Types

The name of a type should be a singular noun that describes a single value of the type.

For example,

stack	**stack_of_integers**
character	**word**
line	**page**

are good type names.

The name of a type should *not* describe the set that is the type, i.e. be a plural noun.

For example, do not use

stacks
characters
words
lines
pages

unless a single value of the type is a collection of individual items.

In this case, however a better name for the type would be the singular name of the type of collection it is, e.g.,

```
set_of_stacks_of_integers
list_of_characters
bunch_of_words
series_of_lines
folio_of_pages
```

Why the insistence on singular nouns?

A singular noun allows you to declare a variable with the type name and to read it aloud properly.

```
Read
    s: stack_of_integers
as
    declare s to be a stack_of_integers
and
    p: page
as
    declare p to be a page
and
    sos: set_of_stacks_of_integers
as
    declare sos to be a set_of_stacks_of_integers
```

Naming of Objects

Ordinary program variables will tend to be short such as the examples on the previous slides, but these variables are *not* exported and would therefore not show up in the module documentation. So we are talking about objects that are important enough to the abstractions that they are exported.

Usually exported objects are constants, but a module itself can be an abstract variable.

For example, in Parnas's KWIC example,

line_storage
circular_shifts
alphabetized_circular_shifts

are all abstract variables for which we must use only exported operations for updating and reading parts of its value.

These names should be nouns that are as descriptive as possible.

The same holds for constants, such as

```
empty_stack
```

in a stack abstraction.

A difficulty is that nouns are used for both the type and objects and one often gets into a situation in which the same noun is reasonable for both a type and a variable of that type.

In such a case, reserve the general noun for the type and use a particularizing adjective or article to make the object, e.g.,

the_stack	
stack_at_hand	the_stack_at_hand
first_page	the_first_page
current_page	the_current_page

Naming Procedures

Procedure names should be imperative sentences with the objects and predicates clearly stated.

For example,

> **push_element_into_stack**
> **sort_input_in_ascending_order_to_produce_output**
> **sort_input_in_nodecreasing_order_to_produce_output**

Note that

    sort_input_to_produce_output

is *not* good because it neglects to mention how the input is being sorted.

There should be a strong enough link between the name and the formal parameter indications to make it clear which parameter is what in the description of the functionality, e.g.,

```
void push_element_into_stack
    (element,*stack)
void push_e_into_s
    (element /*e*/,*stack /*s*/)
procedure push_e_into_s
    (e:element; s:var stack)
```

Naming Functions

Function names should should indicate what value is returned.

For example,

```
get_last_element_that_was_inserted
```

It is not even necessary to make it an imperative sentence, but rather just a description of the value returned, e.g.,

```
the_last_element_that_was_inserted
top_of_stack
```

Doing so allows the function call to be used in the middle of a statement and be readable, e.g.,

```
    t = top_of_stack(s);
```

In the case of a Boolean function, it’s nice to make the name be what is grammatically called a predicate so it can be used after the *if* in a conditional, e.g.

```
is_empty
stack_is_empty
is_last_element
reached_end_of_file
```

so that you can say

```
    if reached_end_of_file(f) then ...
```

Abbreviations

The suggestions above can result in *very* long names and inconvenience for invokers.

Two philosophies:

- **Too bad! It’s for your own good.**
- **OK, make shorter names but use comments.**

For example, in a stack module

```
    void push(*stack_of_integers /*s*/,int /*i*/)
    // push i into s
```

or

```
    void push_i_into_s(*stack_of_integers /*s*/,int /*i*/)
```

but not

```
    void push(*stack_of_integers, int)
```

or

```
    void push_i_into_s(*stack_of_integers, int)
```

the last failing to explain what i and s are.

Examples

Below are two examples of satisfactory documentations for the KWIC example of Parnas.

The first is in Ada and uses short names with commentary.

The second is in C and uses long names with less commentary!

```
package LINE_STORAGE is
-- abstract object providing storage of all
-- of the lines to be indexed

     procedure setchar (l,w,c:INTEGER;
          d:CHARACTER);
--   set the c-th character of the
--   w-th word of the l-th line to d

-- for all l, w, and c, char(l,w,c)
-- is defined only after doing
-- setchar(l,w,c,d) for some d
```

```
     function char (l,w,c:INTEGER)
          return CHARACTER;
--   return the c-th character of the
--   w-th word of the l-th line

-- the following are defined ONLY after
-- setchar been executed to fill the
-- LINE_STORAGE

     function chars (l,w:INTEGER)
          return INTEGER;
--   return the number of characters
--   in the w-th word of the l-th line
```

```
     function words (l:INTEGER)
          return INTEGER;
--   return the number of words in the
--   the l-th line

     function lines return INTEGER;
--   return the number of lines

     -- error handling exceptions
-- SECRETS:
-- The characters are packed four to a
-- machine word, and an otherwise unused
-- character is used to indicate the end
-- of an input word. An index is kept to
-- show the start of each line.

end LINE_STORAGE;
```

```
with LINE_STORAGE;
use LINE_STORAGE;
package CIRCULAR_SHIFTS is
-- abstract object providing storage of
-- all of the circular shifts of the
-- lines to be indexed

     procedure cssetup;
--   initialize CIRCULAR_SHIFTS from
--   LINE_STORAGE

-- the following are defined ONLY after
-- cssetup is done
```

```
      function cschar (l,w,c:INTEGER)
            return CHARACTER;
--    return the c-th character of the
--    w-th word of the l-th cs-line

      function cschars (l,w:INTEGER)
            return INTEGER;
--    return the number of characters
--    in the w-th word of the l-th
--    cs-line
```

```
    function cswords (l:INTEGER)
        return INTEGER;
--  return the number of words in the
--  the l-th cs-line

    function cslines return INTEGER;
--  return the number of cs-lines

    -- error handling exceptions
-- SECRETS:
-- It prepares an index which gives
-- the address of the first character
-- of each circular shift, and the
-- original index of the line in the
-- array made up by module 1. It leaves
-- its output in core with words in pairs
-- (original line number, starting address)

end CIRCULAR_SHIFTS;
```

```
with CIRCULAR_SHIFTS;
use CIRCULAR_SHIFTS;
package ALPHABETIZED_CIRCULAR_SHIFTS is
-- abstract object providing storage of
-- alphabetized list of the circular
-- shifts of the lines to be indexed

     procedure alph;
--   create alphabetized circular
--   shifts from CIRCULAR_SHIFTS
```

```
     function ith(i:INTEGER)
          return INTEGER;
--   index in CIRCULAR_SHIFTS of
--   i-th in alphabetical ordering

     -- error handling exceptions
-- SECRETS:
-- It produces an index in the same
-- form as that produced by
-- CIRCULAR_SHIFTS. In this case,
-- however, the circular shifts are
-- listed in alphabetical order.

end ALPHABETIZED_CIRCULAR_SHIFTS;
```

```
with ALPHABETIZED_CIRCULAR_SHIFTS;
use ALPHABETIZED_CIRCULAR_SHIFTS;
procedure KWIC is
     procedure input is separate;
     procedure output is separate;
     null;
-- does input, alph, and then output
end KWIC;
```

```
with LINE_STORAGE,TEXT_IO;
use LINE_STORAGE,TEXT_IO;
separate(KWIC);
procedure input is
begin
     null;
-- input lines into LINE_STORAGE;
end input;
```

```
with ALPHABETIZED_CIRCULAR_SHIFTS,
    CIRCULAR_SHIFTS,TEXT_IO;
use ALPHABETIZED_CIRCULAR_SHIFTS,
    CIRCULAR_SHIFTS,TEXT_IO;
separate(KWIC);
procedure output is
begin
    null;
-- output ALPHABETIZED_CIRCULAR_SHIFTS
-- in sophisticated way using
-- CIRCULAR_SHIFTS
end output;
```

line_storage.h

```
/* abstract object providing storage of all of the lines
to be indexed */

void set_a_char_in_a_word_of_a_line
     (int /*line_index*/, int /*word_index*/,
     int /*char_index*/, char /*c*/);

/* for all l, w, and c,
     get_a_char_in_a_word_of_a_line(l,w,c)
     is defined only after doing
     set_a_char_in_a_word_of_a_line(l,w,c,d)
     for some d */
```

```
char get_a_char_in_a_word_of_a_line
     (int /*line_index*/, int /*word_index*/,
     int /*char_index*/);

/* the following are defined ONLY after
     set_a_char_in_a_word_of_a_line
     has been executed to fill the
     line_storage */

int no_of_chars_in_a_word_of_a_line
     (int /*line_index*/, int /*word_index*/);
int no_of_words_in_a_line (int /*line_index*/);
int no_of_lines();
```

```
/* SECRETS:
The characters are packed four to a machine
word, and an otherwise unused character is
used to indicate the end of an input word. An
index is kept to show the start of each line. */

/* end LINE_STORAGE */
```

circular_shifts.h

#include line_storage.h
/* abstract object providing storage of all of the circular shifts of the lines to be indexed */

void set_up_circular_shifts_of_lines_from_line_storage();

/* the following are defined ONLY after
    set_up_circular_shifts_of_lines_from_line_storage
    has been executed to fill the circular_shifts */

```
char get_a_char_in_a_word_of_a_cs_line
     (int /*cs_line_index*/, int /*word_index*/,
     int /*char_index*/);
int no_of_chars_in_a_word_of_a_cs_line
     (int /*cs_line_index*/, int /*word_index*/);
int no_of_words_in_a_cs_line (int /*cs_line_index*/);
int no_of_cs_lines();
```

```
/* SECRETS:
It prepares an index which gives the address
of the first character of each circular shift, and
the original index of the line in the array made
up by module 1. It leaves its output in core
with words in pairs (original line number,
starting address) */

/* end CIRCULAR_SHIFTS */
```

alphabetized_circular_shifts.h

```
#include circular_shifts.h
/* abstract object providing storage of alphabetized
list of the circular shifts of the lines to be indexed */

void
    create_alphabetized_circular_shifts_from_circular_shifts
    ();
int
    index_in_circular_shifts_of_ith_in_alphabetical_ordering
    (int i);
```

/* SECRETS:
It produces an index in the same form as that produced by CIRCULAR_SHIFTS. In this case, however, the circular shifts are listed in alphabetical order. */

/* end ALPHABETIZED_CIRCULAR_SHIFTS */

input.h

```
#include line_storage.h
#include <stdio.h>
void input_lines_into_line_storage();
```

output.h

```
#include alphabetized_circular_shifts.h
#include <stdio.h>
void
    output_alphabetized_circular_shifts_in_sophisticated_way
    ();
```

kwic.h

```
#include alphabetized_circular_shifts.h
#include input.h
#include output.h
void main();
```

Global Module

One thing that people do is definitely a No No.

I am referring to a module, often called global.h, that defines problem-related constants that are needed at various places in the program.

The idea is that by putting all of these in the same module, they can all be included by saying only #include "global.h"

For example:

global.h

```
typedef int bool;
#define TRUE 1
#define FALSE 0

#define MAXWORDLENGTH 100
#define MAXLINELENGTH 1024
#define MAXPAGELENGTH 200
```

What is Wrong?

With the definitions of bool, TRUE, and FALSE, nothing. These are not problem-related and they are for getting around restrictions in the programming language.

However, the various MAX__LENGTH constants *are* very much problem related, defining constants needed by the word, line, and page abstractions.

Mainly, they split the abstraction module into two pieces, both of which are needed; someone taking only the word module is surprised to learn that MAXWORDLENGTH is undefined.

Each constant belongs in its own module so that taking the one module takes everything that is needed.

But, what about the convenience of a single #include for all such constants?

The convenience is an illusion.

In any case, one cannot use MAXWORDLENGTH except in a module in which the word module has been included. If MAXWORDLENGTH is defined in the word module, then #include "word" suffices to bring in MAXWORDLENGTH.

Therefore, putting each constant in its own logical module makes the constant visible *only* where it is needed! This is better than with the #include "global.h".

Recording Design Decisions

According to Colin Potts and Glenn Bruns, there are two kinds of design documentation:

- **documentation of the process (deliberation or rationale)**
- **documentation of design results (artifacts)**

Up to now we have discussed only documentation of the design results, showing how the artifacts can be written to be self-documenting with minimal additional comment and describing the necessary additional comments.

Now we consider documenting design rationale.

Why Document Rationale?

Showing rationale shows why the final artifact is the way it is by showing reasons for it being that way and reasons it is not another way.

Without this information, maintainers, when considering alternative ways to achieve particular changes, might repeat the mistakes that were made in the past.

One important class of decisions that should be documented is that of encapsulation itself, why what is encapsulated is and why what is not is not.

Conclusion

So, remember, do what you must to to get those requirements and other documents.

Do not worry if the way you get this information is messy.

Then fake the documentation to look like you got those requirements in a systematic way.