How Precise Documentation allows Information Hiding to Reduce Software Complexity and Increase its Agility"

David Lorge Parnas

When the first papers on “information Hiding” were published (1970-72) , reaction was mixed:

- A (negative) reviewer wrote, “ ••• nobody does it that way” and recommended rejection.
- Fred Brooks called it “a recipe for disaster” (in “Mythical Man Month”).
- Another reviewer called it banal (boring and obvious}.

Ten years later, a textbook discussed those papers saying, “Parnas only wrote down what all good programmers were doing anyway”.

In the 25th anniversary edition of “Mythical Man Month”, Fred Brooks indicated that his original opinion was wrong and wrote “Parnas was right”.

Today, most textbooks indicate that “information hiding” (or related formulations such as structured design and object-orientation) is a good principle but industrial software developers do not do it.

These, obviously contradictory, observations all have part of “the truth” but overlook a basic fact, viz. - if you hide some information, you must give people other information to work with. Information hiding solves many problems but only if the designers pay serious attention to documentation.

This talk reviews the information hiding principle, stating it more precisely than was done when it was introduced, and then illustrates how mathematical documentation can make it work.

Partially supported by the William Mong Engineering Research Fund of The University of Hong Kong

Contents

The Origins of the Information-Hiding Principle

Problems observed in industry (1969)

- Programmers unsure about what they are supposed to do.
- Resulting “modules” do not work together
- Lots of Duplication
- Some gaps,
- Some incompatibilities

My manager asked me how to specify work assignments without writing programs.

- I told him it was easy.
- I tried it and it wasn’t!

Why Was it Hard to Specify Software Work Assignments?

The interfaces were very complex (formats + meaning)/

The real solution would be to simplify the interfaces.

Flash of Understanding: the decomposition (architecture today) was wrong.

Decomposition was based on flow charts.

If the decomposition was based on secrets, we can hide the complexity and not have complex interfaces.

This understanding helped me to understand why Dijkstra’s THE system was “clean” (and to spot the “dirty” parts)

The Napkin of Doom

Compiler and data base experts having working lunch.

They exchange a control block format on a napkin.

I react negatively and ask them to stop.

They try to get rid of me! They “had to communicate.”

I did not know what they should have done.

Napkin is punched, copied, and filed.

Data block format changes but napkin does not.

Components are coupled and don’t work.

Cause of error very hard to find.

Teaching an Early Software Engineering Course

Asked to teach a new “Software Engineering” course.

Nobody could tell me what the content of that course should be or how it should differ from an advanced programming course.

Project with limited information distribution, a “clean” design.

“Module”, a work assignment for an individual or a group.

Project had 5 modules; I divided class into 5 groups of 4.

Each member of a group does one assignment with the same specification

1024 possible combinations, 25 tested and made to work.

Unprecedented! Hard to do today!

Information Hiding is Harder Than it Looks

1. Many have the flowchart instinct; it is what we were taught.
2. Planning for change seems like too much planning
3. Bad role models - We copy old system approaches.
4. Extending bad software
5. Assumptions often go unnoticed
    - Major Flaw in my KWIC Index
6. Reflecting the system environment in the software structure.
7. “Oh, too bad we changed it”.
8. Rush to code.

The Information Hiding Theorem

Information Hiding is often considered an empirical result.

It is a theoretical result!

Theorem: ***If you can prove program A correct knowing only the interface to program B, then if you change B without changing its interface, A need not change (proof still valid).***

Requires empirical verification

- **That people can know what to hide and what they are hiding.**
- **That the interfaces can be efficient**
- **That we can describe interfaces without revealing secrets.**

The First Reactions

“IBM doesn’t do it that way”. (My colleagues at work!)

“Parnas obviously doesn’t know what he is talking about because nobody does it that way” (Referee rejecting paper)

“A Recipe For Disaster” (Fred Brooks-Mythical Man Month)

Later Reactions

10 years later: “Parnas only wrote down what all good programmers were doing anyway”. (SE book)

25 years later “Parnas was right!” (Fred Brooks, revised MMM)

Many times: “It’s not enough, we still write bad software”. (many software researchers with other ideas).

What’s the Truth?

There were no good programmers (smile).

Fred Brooks was right both times:

- Information Hiding was the second of two papers.
- The first was on interface specification.
- The second without the first would be a disaster.

We still have bad software because Information Hiding is not done well.

- Secrets show (even in textbook examples).
- The abstraction is not documented.

Designing the Module Structure

The most important structure

- Constrains all of the others
- Essential before implementation can proceed.

What would make a module structure good?

- Parts can be designed independently.
- Parts can be implemented independently.
- There are no duplicate or almost alike pieces of code (clones, cut/pasties)
- Parts can be tested independently.
- Parts can be changed independently.
- Integration is easy if the modules met their specification
- Maintainers can find the relevant code and change it quickly

Modules vs. Components

Module (historically):

- Task (work-assignments) for developers (management unit).
- Should have a “secret” that it hides. Only module that has to be changed if the hidden decision has to be changed.

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 interface documentation scheme for both.

Designing the Module Structure

What are the principles?

- Information Hiding (intended to sound subversive, different).
- Abstract Data Types (sounds good, vague)
- Separation of Concerns (motivation)
- Object Orientation (emphasis on what is created, false emphasis on language)
- Component Oriented Software (new word for old idea, concepts)

These are all the same design principle.

- People focus on the syntax and buzzwords and forget the design principle.
- Newer systems often look just like the old ones if you ignore the syntax.
- Ask designers to tell you the secret and they usually cannot.
- Review reveals distributed secrets and secret-free modules.
- After a conscious review of the secrets, the structure is always better.

Larger Systems

There are many implementation decisions
(hundreds, hard to count).

There are many details.

The structure is inherently much more complex.

This leads to new issues. Among them:

- How can we keep the project under intellectual control?
- How can we maintain conceptual integrity?
- How can we keep the maintenance cost down?
- How do we deal with unstructured lists of modules?
- How can we tell when we have them all?
- How does everyone remember the names?
- How do we avoid duplication?•

Group Modules Into Classes

Apply the principle of refinement (partition)

- Eases check for completeness
- Assures lack of duplication (if done properly)
- Eases achievement of consistency.

Leads to more helpful naming conventions.

Make a specific module easier to find.

The SCR A-7E Module Structure

Top-level decomposition

1. Support System-hiding module (hardware and software)
2. Behaviour - hiding module
3. Software decision module

If the secret is in the software requirements document, it must be (1) or (2).

If it is not a requirement, it must be (3).

This “top level” structure appears to be universal.

The SCR A-7E Module Structure
Second-level decomposition

1. Hardware-hiding module decomposition

    1.1 Extended computer module (virtual machine and OS)

    1.2 Device interface module

Note: If it affects more than one device, consider it part of the computer.

The SCR A-7E Module Structure
Third-level Decomposition

2. Device interface module decomposition

- 2.1 Air data computer
- 2.2 Angle of attack sensor
- 2.3 Audible signal device
- 2.4 Computer fail device
- 2.5 Doppler radar set
- 2.6 Flight information displays
- 2.7 Forward looking radar
- 2.8 Head-up display (HUD)
- 2.9 Inertial measurement set (IMS/IMU)
- 2.10 Panel

and another dozen modules.

Contrast abstract device interface with the “termcap” approach.

Documentation of a Module Structure

Module structures are documented in a *Module Guide*, which states only the secret of each module.

- **The module guide is hierarchically structured; the structure of the document mirrors that of the system.**
- **The top level partitions the system into modules.**
- **Any module may be partitioned (if secrets can change independently)**
- **Partitioning introduces no new secrets: refines the old ones.**
- **There should be no shared secrets.**
- **Stop partitioning only if a module is easily replaced.**

The module guide is the only document read by all

It is substituted by per-module documents.

Aside: There are no “Levels of Abstraction”.

Some authors and teachers use the phrase “levels of abstraction”

Terminology seems innocent but is not clearly defined.

The term “levels” is used properly only if one has defined a relation that is loop free.

None of the authors has defined the relation, “more abstract than”,

Not all programs in an information-hiding work assignment need to be at the same level.

Many examples are actually quite wrong. (e.g. ISO 7-layer model)

The Message: Document Document Document

Failure to carefully apply information hiding and to document well causes the problems that we have with software.

- Precise complete requirements documents would constrain the programmer properly.
- Documentation can reduce the “mythical man month problem”.
- Information hiding reduces the “ripple effect”
- Documentation permit design reviews.
- Documentation support testing.
- Documentation make inspection more effective.
- Documentation make maintenance easier.

The question is “Can we produce better documents?

- **The answer is “Yes, we can!”.**

What Is Meant by “Document”

A record of design decisions that is binding, i.e. a restriction of future decisions.

To be as useful as possible these documents must be:

- Accurate
- Consistent
- Complete (all required decisions fully documented).

Informal introductions/explanations are not documents in this sense.

Steps Towards Better Documents

Use mathematics to get precision.

Use mathematics to make checking and testing tools possible.

Use tabular expressions to get readability and structure.

Use rigid documentation rules to make it easy to find the information. (Information retrieval system).

Build tools to make useful documents even more useful.

Build tools to make it easier to produce good documents.

A Preliminary Example: Dell Keyboard Checker

In daily use in Limerick for many years.

Believed to be completely correct.

Two documents totaling 21 pages (English).

- ambiguities
- missing cases
- errors

Posed as a challenge by skeptical manager

All information could be expressed in one page

- revealed errors in program and documents
- much more precise and easily used.
- served as input to testing and inspection.

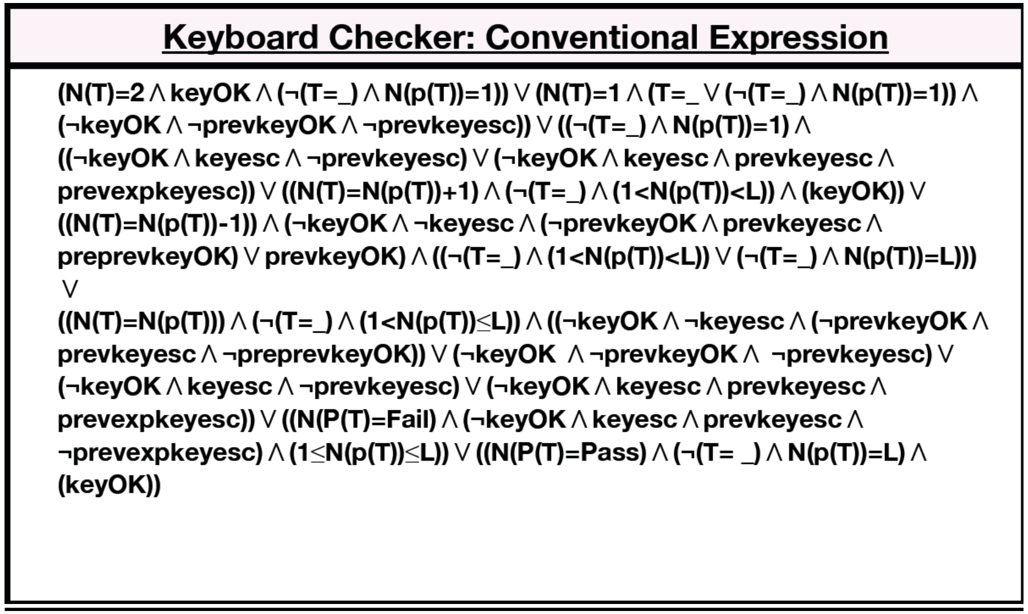
Keyboard Checker: Tabular Expression

N(T) =

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

Requirements for Keyboard Checker

No Theoretical Advantage



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))

Just practical advantages:

- fewer errors
- checkability
- ease of reference.

Documents Are Not Programs

They must describe mappings from input to output.

They must not describe the steps in a computation process.

They must not provide information that should not be in the document[1].

They must provide the information that the intended readership needs

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

[1] Content definitions for documents will be discussed later.

Practical Experience (30 years)

US military aircraft (project still ongoing)

- Answer any requirements question looking at ≤ 7 pages.

Nuclear Plants (ongoing)

- 218 errors found in code that had been tested for six years.
- no further errors found in more than 15 years of use.

Management Information Systems for Bell Labs. (?)

- 14 “copycat” projects in first year
- shortest on-site testing time on record

Mobile Telephone connection software (base station)

- collected information that was distributed around the world into one easily referenced document.

The Role of Documents in Traditional Engineering

Engineers design through documentation

Documents record key design decisions

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

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”.

More About What I Mean by "Documentation"?

Practical tool, not just a theoretical achievement and a demonstration of diligence.

They must have the following properties.

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

What We Must Do To Improve Software Documents

Accuracy, precision, and consistency can be improved by using mathematics.

Consistency, completeness, ease of reference can be improved by tabular notation and rigid organization.

Consistency and ease of finding information are improved by having content definitions for each document.

Some 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* contains 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.

- This has confused many researchers.
- These classifications are a matter of intent not notation.
- There is no such thing as a “specification language”.

Documents vs. Models

Two Questions:

- Are models documents? (not usually)
- Are documents models? (yes, good ones!)

What is a model?

- A simplified version of “the real thing” that is easier to study.
- Models have some properties of the “real thing”
- Not all properties of the models are properties of the “real thing”.

Models are not usually descriptions.

- Everything you can learn from a description must be true of reality.

What you can learn from a model of a Boeing 747?

- That it has a bump, two big wings, two small wings ...
- That it is plastic, has painted “windows”, can be held in your hand, …

Documents are models that you can trust.

Is “Mathematical Documentation” = “Formal Methods”?

The differences:

- **Information organized for easy retrieval.**
- **No new mathematics - classical concepts suffice.**
- **Not modeled on philosophers’ view of logic**
- **Engineering style of mathematics (“closed-form” solutions)**
- **Not axiomatic in style - readers evaluate, not derive**
- **Not intended for automatic verification (possible *in theory*).**
- **Proof is not the main goal (but possible *in theory*).**
- **Documents are not programs**
- **Attention to authors and audiences - who needs what information.**

Is This “Formal Methods”?

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

How about this?

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

The above is a tabular mathematical expression.

This Too is A Mathematical Expression

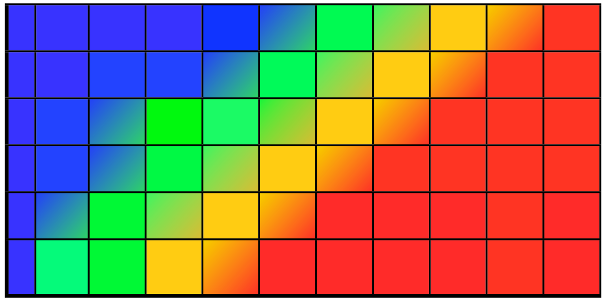
BMI

T[1]

p<88	88≤p<110	110≤p<121	121≤p<132	132≤p<154	154≤p<176	176≤p<198	198≤p<220	220≤p<110	242≤p<265	p>265

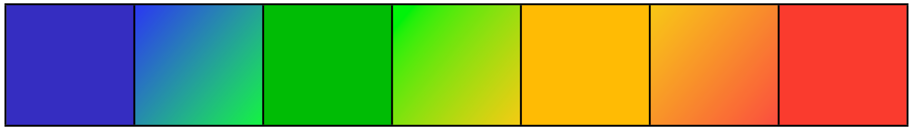
T[4]	T[2]
1.9≤m<2	very talll
1.8≤m<1.9	talll
1.7≤m<1.8	avg
1.6≤m<1.7	middle
1.5≤m<1.6	short
m≤1.5	very short

T[0]



T[3]

k<40	40≤k<50	50≤k<55	55≤k<60	60≤k<70	70≤k<80	80≤k<90	90≤k<100	100≤k<110	110≤k<120	k>120



underweight	low borderline	normal	low overweight	overweight	very overweight	**OBESE!**

Interfaces

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

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

Definition: Interface

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

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

Interfaces determine difficulty of changing software.

Surprising Observations about Interfaces.

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

Interface Design

Since interface changes affect more than one module, care is required.

The assumptions in an interface should be unlikely to change.

Since software designers may not know what will change, the assumptions must be documented and reviewed by the relevant experts.

Assumptions about what information will be made available by a component and what information will be needed by a component are less likely to change than assumptions about the representation of such information.

Interfaces in the form of get/set programs are less likely to change and the changes are often only additions.

Assumptions are often very subtle.

Changes are often only predictable by specialists.

Because the software developers often don’t know the application details and the people who do don’t understand software, a disciplined systematic design and review procedure is needed.

Interface Documentation

The most important interfaces are those between modules and between components.

If these are information-hiding they must be described using a “black box” method.

Much of the internal state representation will be hidden.

Trace Function Method analogous to SCR method.

Value of each output is shown as function of history of inputs and outputs.

Extract from Module Interface Document - I

Time Storage Module

Output Variables

Variable Name	Type
hr	<integer>
min	<integer>

Access Programs

Program Name	‘in	Abbreviated Event Descriptor
SET HR	<integer>	(**PGM**:SET HR, ‘in, hr’)
SET MIN	<integer>	(**PGM**: SET MIN, ‘in, min’)
INC		(**PGM**:INC, hr’, min’)
DEC		(**PGM**:DEC, hr’, min’)

Extract from Module Interface Document - II

Output Functions

hr(T) ≡

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

The Bottom Lines:

Producing no documentation gets you in trouble.

Producing bad documentation might be worse.

Producing good documentation:

- will help you get the requirements right
- will help you get interfaces right
- will help you in your testing
- will help you in your inspections
- will help you in maintenance and upgrades
- will help you manage a product line effectively.

Management’s Role in Document Driven Design

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

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

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

Without management support it won’t work!

Conclusions

Information-Hiding is a simple mathematical theorem.

Information-Hiding Works but it is not enough.

One needs precise documentation to avoid Brooks’ disaster.

Mathematical Documentation is the way to make information-hiding work and to make software better.

Its not an easy solution but it is a solution.