

Document Driven Disciplined Development of Software

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)
University of Limerick,¹ Ireland

It is no accident that the branches of Engineering are called “disciplines”. Every properly educated Engineer has learned that the design of quality products requires discipline and adherence to standard procedures. Engineers understand that they must produce a specified set of documents and perform a variety of analyses whose results must be included in the documents. Engineers who do these things are less likely to produce a defective product. In many jurisdictions, engineers who fail to follow the standard discipline may be considered to have been legally negligent. Software development should not be different but it is.

Document standards can help to support and enforce the appropriate discipline. We describe a set of documents that contain the information required for disciplined development. A developer who completes these documents properly will have performed the analyses necessary to assure that the product will be of high quality. This talk does not describe a “process”. Discipline does not require that these documents be completed in any particular order - only that they are all eventually completed properly.

¹ On leave from McMaster University in Hamilton Ontario Canada



University of Limerick

Preamble (Midweek Sermon)

When the going gets tough, Computer Science runs away.

- “Program Verification” is no longer a problem of interest. (i.e. not popular)
- “Non Functional Requirements”

Those who ignore the past are doomed to reinvent it.

- “What’s the IBM 7090 Linking Loader?” (inventor of something similar)
- “At Sun we don’t pay much attention to what IBMers say” (Java expert)
- “Aspects are different from concerns in some way” (Aspect speaker)

Those who lose touch with reality will be found irrelevant and forgotten.

- “I don’t know anyone who uses Verilog” (Author of ‘n’ Verilog semantics)
- “Publication is a stricter test than a reality check” (SFI referee)

Renaming the problem lets you start over (from zero).

- “Didn’t read about program families - product lines author.
- separation of concerns, modularity, components, aspects ...



The Old Problems are Still Real. The Real Problems are Old

Specifications and other forms of software documentation were real problems in the 50s. They are real problems today.

Pre/Post conditions have not solved the problem.

So called “Formal Methods” have only raised new problems.

It is easy to be theoretically correct.

You can get anything published if you keep resubmitting.

Getting something that works in practice is what’s hard.

We should still be working on many of the problems of the 1950s.

The vast majority of software problems that I see are either caused or exacerbated by a lack of precise, complete, and correct design documentation.

Documentation is a problem that is both real and old.



University of Limerick

Outline

- (1) Disciplined design through documentation
- (2) Requirements identification and documentation
- (3) Module interface design and documentation
- (4) Module internal design and documentation
- (5) Designing and documenting the “program uses program” relation
- (6) Documentation based disciplined quality assessment (inspection, testing)
- (7) Where do we get the time?



The Role of Discipline in Engineering

Engineers are taught to produce high quality products whose properties are completely understood. Surprises and oversights are considered failures.

Disciplined application of sound science, using the necessary mathematics, is the essence of Professional Engineering.

For each Engineering discipline there are standard documents, standard analyses, standard metrics, standard checklists.

In software development, surprises and oversights are considered normal “bugs”. This is not caused by a theory deficit; the poor quality is associated with oversights that, in most cases, could have been eliminated by a disciplined analysis.

Some software developers consider themselves creative artists! They resent any rules or restrictions. They resent being asked to document.

An improvement in software quality requires that we show people how to do what is required, make it easier to do those things, and then require that they do it.



University of Limerick

Discipline vs. Process

Exercising discipline does not dictate a process.

Process dictates the *sequence* of decisions.

Discipline requires that the *final product* be systematically documented, carefully verified, and thoroughly tested.

There are many reasons why we cannot follow the most rational process¹.

There is no excuse for not exercising the required discipline.

¹ Parnas, D.L., Clements, P.C., “A Rational Design Process: How and Why to Fake It”, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986



Disciplined Design through Documentation

There is no magic easy way to high quality. It requires discipline!

Documentation can make discipline easier.

Documentation can make discipline easier to manage.

We cannot tell people how to think; we can make them document the results of their thoughts in a disciplined, checkable way.

The road to better software stands on a foundation of documents that are **actually used**:

- to record design decisions
- to review design decisions
- to guide future design decisions and revisions

The documents must be (1) written to meet standards, (2) reviewed against standards, and (3) be fully taken into account in future development.



How Can Documents Support Disciplined Design.

Most of us are (all too) familiar with tax forms.

Computing your income tax requires discipline to be sure that you take all factors into account and perform all steps properly.

The return form, helps you to do that.

You are not restricted in the order in which you enter information, but, you must eventually enter it all and use it all.

Software documentation standardised forms can play the same role.

- they tell you what information must be included
- they tell you how to organize it
- they tell you how to check for completeness and consistency.



The First Step - Defining Document Content

There are endless discussions about the content of each document; standards usually tell you the format and structure but only sketch the content.

One paper, defines the content precisely:

Parnas, D.L., Madey, J., “Functional Documentation for Computer Systems Engineering”, Science of Computer Programming (Elsevier) vol. 25, number 1, October 1995, pp 41-61

This (very abstract) paper tells you what should be in each document. It does not say how to format the information or what notation to use.

It does not give you section headings. This is different from most documentation standards.

Each document must give the representation of a specific relation.

Disciplined development requires completing these documents and checking them against their definition.



How can Documents be Both Precise and Readable?

This is precise:

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

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.

$$\begin{aligned} &(((\exists i, B[i] = 'x) \wedge (B[j'] = x) \wedge (\text{present}' = \mathbf{true})) \vee ((\forall i, ((1 \leq i \leq N) \Rightarrow B[i] \neq x)) \wedge (\text{present}' = \mathbf{false}))) \wedge ('x = x' \wedge 'B = B') \\ &(((\exists i, B[i] = 'x) \wedge (B[j'] = x) \wedge (\text{present}' = \mathbf{true})) \vee ((\forall i, ((1 \leq i \leq N) \Rightarrow B[i] \neq x)) \wedge (\text{present}' = \mathbf{false}))) \\ &(((\exists i, B[i] = 'x) \wedge (B[j'] = x) \wedge (\text{present}' = \mathbf{true})) \vee ((\forall i, ((1 \leq i \leq N) \Rightarrow B[i] \neq x)) \wedge (\text{present}' = \mathbf{false}))) \end{aligned}$$

$$(\forall i, ((1 \leq i \leq N) \Rightarrow B[i] \neq x))$$

It is not more formal or difficult than a programming language but parsing is difficult.



How can Documents be Both Precise and Readable?

The following 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 more precise than the text above:

Specification for a search program

| | | |
|------------------|-----------------------------------|-------------------------|
| | x can be found in B | x can not be found in B |
| j' must be | a place where x can be found in B | any number at all |
| present' must be | true | false |



How can Documents be both Precise and Readable?

This is precise and readable (by trained people).

Specification for a search program

| | | | |
|------------|---------------------------|-------------------------------|-------------------|
| | $(\exists i, B[i] = x)^a$ | $(\forall i, \neg(B[i] = x))$ | |
| $j' \mid$ | $B[j'] = x$ | <i>true</i> | $\wedge NC(x, B)$ |
| present' = | true | false | |

a. These tables depend on using a logic in which evaluating a partial function outside of its domain yields “*false*” for all built-in predicates.

- 1 The first can be input to mathematics based tools but is hard 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.

The table parses the expression for the reader, but is still mathematics.



Tabular Notation Is Useful For All Software Design Documents

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

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

These documents can be tested for completeness and consistency.

Producing these documents is part of “the discipline”.

Checking them is the remainder of the discipline.

They make the programming *much faster* and *more reliable*.

But, most important:

They can be read by human beings.

- They were read and corrected by pilots, engineers, telephone operators, and even managers.

They help us to reduce the number of errors.



Incomplete and Imperfect Documents

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

With software, “imperfection” is not theoretically 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. Relational models include it automatically.

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

- This is a red herring thrown out by those who do not want to change their ways.



Requirements Documentation - Why?

Programmers should not make decisions that are visible to users.

- User visible external characteristics of the system should be determined, documented, and reviewed by those who will have to use and sell the product, not by coders. Good coders may not have the necessary information.
- Decisions about hardware interfaces should be made and documented by experts in the devices.

Where external characteristics cannot be determined in advance, or may change after development, programmers should be told to prepare for those changes.

The requirements document communicates this information to the programmers.

Requirements documents can later be used to generate real-time monitors and test oracles. They are very valuable in software inspections.



Requirements Documentation - The Basis

Requirements can not be adequately documented as a wish list.

There must be a complete list of all *controlled variables*

There must be a complete list of all *monitored variables*.

The value of each controlled variable at any time must be specified as a (mathematical) function of the history of the values of the monitored variables.

These are essential for requirements statements that leave nothing but implementation decisions to the programmer.

In practice, the lists change while the system is being developed but a first draft is the best starting point.

Changeability is also a requirement, but one must specify classes of changes. We cannot design systems in which everything is equally easy to change.



Mathematical Definition of Document Contents

The implementors need to know the following relations:

Relation NAT:

- domain contains values of $\underline{\mathbf{m}}^t$, range contains values of $\underline{\mathbf{c}}^t$,
- $(\underline{\mathbf{m}}^t, \underline{\mathbf{c}}^t)$ 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 $\underline{\mathbf{m}}^t$, range contains values of $\underline{\mathbf{c}}^t$,
- $(\underline{\mathbf{m}}^t, \underline{\mathbf{c}}^t)$ 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. This (control theoretic) model underlies all of our examples.



Disciplined Documentation of Requirements

Talking to users is essential but far from enough.

We need a disciplined interview process.

- Identify all of the controlled (output) variables.
- Identify all of the observed/measured (input) variables.
- Identify how the input variables can change over time (maximum, minimum, rate of change, time sequence of values, etc.)
- Document any relationship between these values that exists without the product.
- Document the restricted relationship that should exist because of the product.

This information should be the product of requirements identification. When it is not available, the possibilities should be identified.

“Don’t leave “home” without it!”



University of Limerick

Disciplined Design Documentation

In Engineering we document a design before we start to build.

In Software Engineering we should do the same.

Designs should be reviewed carefully before investing in implementation.

If we do this:

- The coding and subsequent testing will go faster.
- The quality will be higher.
- Change and correction will be easier.



Module Interface Design and Documentation

Every interface corresponds to a set of assumptions.

If those assumptions are false, or become false later, “ripple effect” changes can be expected.

The assumptions should be clearly stated and reviewed so that we all know what changes will affect more than one component.

Most software interfaces today are *ad hoc*; the assumptions are unconscious, unintended, unreviewed, and undocumented.

This reflects a lack of design discipline for interfaces and lack of a disciplined review of those interfaces.



University of Limerick

What is an Interface?

The interface between two programs consists of all the assumptions the developers of each program made about the other.

- More than just syntax, format, and type information.
- If one component of a system makes an assumption about another component of the system, and a change in the second component makes that assumption false, the first component will have to be changed.

In other words:

- If the correctness of one component of a system can only be demonstrated by making an assumption about another component of the system, and a change in the other component makes that assumption false, the first component will have to be changed.
- If we want to avoid the “ripple effect”, we must be aware of the assumptions implicit in an interface and design interfaces that are unlikely to change.



University of Limerick

Designing Abstract Interfaces

What is meant by “abstract”?

- Not vague, theoretical or highly mathematical; abstract means - expressing a general property.
- Abstract implies a **many-to-one** mapping.
- The abstraction represents many things equally well.
- The abstraction models some aspects of the real things, but not all.
- Eliminating detail is the approach: The interesting issue is *which* details should be eliminated. Eliminate the details that are likely to change.

Examples of abstractions:

- Circuit diagrams - Physical layout is hidden.
- Graphs - Meaning of nodes and arcs is hidden.
- Algorithms - abstract from restrictions of programming languages.
- Abstract Data types - Representation is hidden.



University of Limerick

Abstractions

Why are abstractions useful?

- If all properties of the abstract system correspond to properties of the real system, we can learn about the real system by studying the abstraction.
- Abstraction is simpler (in principle);
 - It has less information, but
 - It may appear more complex because it is described using unfamiliar notation.
- Results about abstraction may be “reused”.
 - They apply in many situations - those situations that share the abstraction and differ only in things that are abstracted from.



An Abstraction is Different from a Lie!

An abstraction must be valid for many real things.

A lie is true for nothing.

Many assumptions commonly identified as abstractions are really lies:

- An infinite memory is not an abstraction; it is a lie.
- Infinite speed is not an abstraction; it is a lie.
- Size and speed can be parameters; this is an abstraction - not necessarily a lie.
- Sometimes the assumption of a single parameter is itself a lie.

Interface documents must be true abstractions.



University of Limerick

Abstract Interfaces

What is an abstract interface?

- One interface that represents many actual interfaces equally well.
- An interface that models some properties of actual interface but not all.
- A (proper) subset of the set of assumptions in the actual interface.

All of the assumptions must be true.

- It will not apply to all stacks if you assume that the parameter's value must be a constant independent of the values stored in the stack.

Abstractions can introduce restrictions but do so consciously.



Implementation Using an Abstract Interface

The implementer of the interface uses the specification and the concrete interface. They use only the specification and no information about the application.

The implementers of the programs that use the abstract interface use only their knowledge of the application (another specification) and the specification of the abstract interface. They have no information about the concrete interface.



University of Limerick

Interface Specification Discipline

Interfaces between components must be carefully specified in a precise and testable way.

- We should be able to easily identify all the outputs of a component.
- We should be able to easily identify all the inputs of a component.
- We should be able to describe the value of any output as a function¹ of the history of the inputs.

If you cannot do these things, you do not know your product and it will be full of surprises.

The specification should be so complete and readable that only those responsible for its implementation want to look at the code.

The specification should be so complete and readable that those responsible for its implementation need not ask how it will be used.

¹ In some circumstances the relation between the output value and the history of the inputs may not be a function.



A “Black Box” Module Specification

N(T)=

| | | | |
|--------------|-------------------------------|-------------------|-------------------|
| | $\neg(T = _) \wedge$ | | |
| T = _ | 1 < N(pr(T)) < L | N(pr(T))=L | N(pr(T))=1 |

| |
|--|
| $nt(T) = N(pr(T))$ |
| $\neg(nt(T) = N(pr(T))) \wedge \neg(nt(T) = esc)$ |
| $\neg(nt(T) = N(pr(T))) \wedge (nt(T) = esc) \wedge \neg(nt(pr(T)) = esc)$ |
| $\neg (nt(T) = N(pr(T))) \wedge (nt(T) = esc) \wedge (nt(pr(T)) = esc)$ |

| | | | |
|---|------------------|------------------|---|
| | $N(pr(1,T)) + 1$ | | 2 |
| 1 | $N(pr(1,T)) - 1$ | $N(pr(1,T)) - 1$ | 1 |
| | $N(pr(1,T))$ | $N(pr(1,T))$ | 1 |
| | | | |

Status(T)=

| | | | |
|--------------|-------------------------------|---------------------|---------------------|
| | $\neg(T = _) \wedge$ | | |
| T = _ | 1 < N(pr(T)) < L | N(pr(1,T))=L | N(pr(1,T))=1 |

| |
|--|
| $nt(T) = N(pr(T))$ |
| $\neg(nt(T) = N(pr(T))) \wedge \neg(nt(T) = esc)$ |
| $\neg(nt(T) = N(pr(T))) \wedge (nt(T) = esc) \wedge \neg(nt(pr(T)) = esc)$ |
| $\neg (nt(T) = N(pr(T))) \wedge (nt(T) = esc) \wedge (nt(pr(T)) = esc)$ |

| | | | |
|--------------|--------------|--------------|--------------|
| | “incomplete” | “pass” | incomplete |
| “incomplete” | “incomplete” | “incomplete” | “incomplete” |
| | “incomplete” | “incomplete” | “incomplete” |
| | “fail” | “fail” | “fail” |

T is the input history. No internal structures are mentioned.



Module Internal Design and Documentation

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 (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 functions” mapping from concrete states to concrete states.
- Such functions are often best described by tabular notation.



University of Limerick

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:

$edge \triangleq (R = F + 1) \vee (F = QSIZE - 1) \wedge (R = 0) \triangleq$

$\langle qs \rangle \triangleq qds \times 0..QSIZE - 1 \times 0..QSIZE - 1 \times boolean$



University of Limerick

The Abstraction Function¹

af: $\langle qs \rangle \rightarrow \langle queue12 \rangle$

af(DATA,F,R,FULL,old) \Leftarrow

| | |
|--|---|
| $(\neg edge \vee FULL) \wedge (F \geq R) \wedge old$ | Q12INIT.(DATA[F]).(DATA[F-1]). ... (DATA[R]) |
| $(\neg edge \vee FULL) \wedge (F < R) \wedge old$ | Q12INIT.(DATA[F]).(DATA[0]).(DATA[QSIZE-1]).(DATA[R]) |
| $edge \wedge \neg FULL \wedge old$ | Q12INIT |
| $\neg old$ | |

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,
- *specifications*, which describe the requirements that a product must meet, and
- *models*, simplified descriptions which have properties that the original does not have. (In other words, dangerous lies).

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 (LD-relations).
- 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).



University of Limerick

Queue: Program Functions

pf_Q12INIT \Leftarrow

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

gpf_ADD(a) \Leftarrow NC(F) \wedge $\forall j$ ($j \neq R'$) [NC(DATA[j])] \wedge NC(a) \wedge

| | ('R = 0) \wedge old \wedge | | | ('R \neq 0) \wedge old \wedge | | | \neg old |
|------------|--------------------------------|--------------|--------------|-------------------------------------|--------------|--------------|------------|
| | 'edge \wedge | | \neg 'edge | 'edge \wedge | | \neg 'edge | |
| | 'FULL | \neg 'FULL | | 'FULL | \neg '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 \Leftarrow NC(DATA,R) \wedge

| | $(\neg$ 'edge \vee 'FULL) \wedge old \wedge | | $($ 'edge \wedge \neg 'FULL) \vee \neg old |
|---------|---|----------|--|
| | ('F = 0) | ('F > 0) | |
| F' = | QSIZE-1 | 'F - 1 | 'F |
| FULL' = | false | false | 'FULL |

pf_FRONT \Leftarrow NC(R,FULL, DATA, F) \wedge

| | \neg 'edge \vee 'FULL old \wedge | $($ 'edge \wedge \neg 'FULL) \vee \neg old |
|----------------|--|--|
| return value = | 'DATA['F] | |



What Good are these Documents?

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.

No time is lost.

The tables make the language independent design decisions.

The tables are easy reference material.

The second table also shows how code can be simplified.

Test oracles can be generated.

Other testing functions (e.g. coverage measures) are supported.

These tables support a systematic inspection process.

They take effort but that effort is not wasted because they are used repeatedly in subsequent work.



University of Limerick

Checking A Module Design

A module design may be wrong, i.e even if the programs that implement it are satisfy their specifications, the module 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.

We can check before implementation that the design can be made to work.

There is an equation that should be verified.

We always make mistakes along the way but we can do better.



Designing and Documenting the “Program Uses Program”

Note that there are many programs in a module.

Definition of uses:

- Given program A with specification S and program B, we say that A uses B if A cannot satisfy S unless B is present and functioning correctly.
- Example: hardware for division use power supply but calls divide by 0 routine

Virtual-machine analogy

This determines what subsets are executable.

Extensibility depends on this (and interface design) as well.

This is often a programmer’s casual decision; it should be designed and reviewed before programming.

Not all programs in a module need be on the same level.

This is not a “module uses module” hierarchy.



Systematic Quality Assessment (Inspection, Testing)

Even the best software seems to have bugs and quirks.

Systematic testing (not “try it you’ll like it) and disciplined inspection can find the bugs *before* they are released.

Software is tested against the design documents.

Design documents also used (or produced!) during inspection.

Experience (Darlington Nuclear plant):

- 218 discrepancies found after 6 years of better than average automated testing. (200 were not considered errors.)
- 15 years of active use and change with only one error (analysis) found



University of Limerick

Code Inspection Discipline

The secret of inspection is “divide and conquer”.

Each part must be small enough to be completely inspected.

Each inspection must consider every variable and every case.

There must be a method that insures that each separately inspected part fits with the others and that no part has been overlooked.

This requires discipline and strict methods.

If you don't do this, you cannot trust the results of your inspection.

If you design and document using displays, inspection is quicker and more effective.



University of Limerick

The Concept of Displays

A display consists of three parts:

- (1) A specification describing what the program should do.
- (2) the program itself.
- (3) specifications of subprograms invoked by this program.

All programs in part 2 can be short because they invoke other programs.

A display presents a program in such a way that its correctness can be examined without looking at any other displays.

A set of displays is complete if, for each specification of a non-standard subprogram found in part (3), there is exactly one display in which this specification forms part (1). Completeness can be checked mechanically.

A display is correct if the program in part (2) will satisfy the specification in part (1) provided that all invoked programs satisfy the specifications in part (3).

A set of displays is correct if complete and all displays correct.



University of Limerick

SQRL Code Inspection Process

- (1) Prepare a precise specification of what code should do.
- (2) Decompose the program hierarchically.
- (3) Produce the descriptions required for the “display approach”.
- (4) Compare the “top level” display description with the requirement specification.

Observations:

- You can't inspect without precise requirements.
- Step (2) would already be done if you use the display method for documentation.
- Step (3) is truly an active design review
- All reviewer work is itself reviewable.
- **If you did not already have it, the by-product is thorough documentation.**
- It's a systematic sequence of small steps.



What are the Limits of Testing and Inspection/Proof?

“Testing can show the presence of bugs but never their absence.”
(E.W. Dijkstra)

- False in theory, but true in practice.

In most cases it is impractical to use testing to demonstrate trustworthiness.

One *can* use testing to assess reliability.

Verification and Inspection are based on assumptions about the support system and environment. They will not reveal false assumptions.

Two sides of a coin:

- I would not trust an untested program!
- At Darlington we found serious errors in safety-critical programs that had been tested for years!



University of Limerick

Testing Discipline

Testing cannot be replaced by inspection or verification.

- Testing and inspection are complementary.
- Both can be based on the same documents.
 - Documents can be used to generate oracles
 - Documents can be used to measure test coverage
 - Documents can be used to generate test data.

Coverage will (almost) **always** be inadequate, but

- . . . it can be better than it usually is
- ... it can be designed to stress an implementation
- . . . it can be statistically significant for reliability prediction.

The hardest questions remain, “What do you do if all tests are passed? What do you know if all tests are passed? When have you tested enough?”



Why is Software Hard to Test?

1. It has no natural internal boundaries.
2. It is sensitive to minor errors - there is no meaning to “almost right”. (chaotic behaviour).
3. There are too many cases to test completely.
4. There is no natural structure to the space of test cases.
5. Interpolation is not valid.
6. There are “sleeper bugs”, making the testing of real-time software much more difficult. (No “time constants”) How long must a single test be?

These are “inherent” properties, not signs of immaturity in the field.
These are old problems and eternal problems.



University of Limerick

Three Kinds of Testing

Black Box Testing

- Testing is based on specification alone.
- Cases are chosen without looking at code.
- When testing an abstractly specified (information hiding) module you must not use information about the data structure.

Clear Box Testing

- Test choices based on code and data structure.
- Use code coverage criteria such as those described earlier.

Grey Box Testing

- Take advantage of knowledge about data structure but not program structure.
- Unavoidable for modules with memory - testing depends on number of states.

The Documentation we have been discussing is an essential input to Black and Grey Box testing and helpful for Clear Box testing.



How can Precise Design Documents Help Testers?

Having documentation ameliorates the following problems.

- Component testing requires precise definition of component boundaries.
- Statistical data is needed for reliability estimation of components.
- Some components may be ready for testing before others.
- Generating test cases is prone to oversight and very time consuming.
- Evaluating test results is very time consuming and prone to oversight.
- Difficult to generate test cases until program is complete.
- Difficult to evaluate thoroughness of test coverage.

One of the ways that the time spent on documents is paid back is faster, more effective, testing.



What's Wrong With "Formal Methods"

- (1) Unnecessary word - just applying mathematics
- (2) Based on logicians, philosophers not engineers, applied mathematics.
- (3) Do not deal with complexity
- (4) Do not summarise behaviour (most cases)

Is this really different?

- Yes, relational/functional model
- Yes, Different Models for different complementary documents
- Yes, complete documentation (including error cases) not models
- Yes, summary/abstraction in every document
- Yes, realistic notation.



University of Limerick

Where do We Get the Time?

No less an authority than Dilbert, told that he should work as an Engineer, responded with, “As if we had time for that.”

Here is what we really do not have time for:

- Testers having to document the requirements to prepare their tests.
- Reviewers having to search for requirements information
- Maintainers trying to reconstruct interfaces that were not documented
- Maintainers trying to figure out what the data structures mean.
- Maintainers trying to figure out what programs must be present for their program to run.
- Designers guessing requirements that were not specified.



University of Limerick

Conclusions

“Popularity” is not a criteria for choosing problems or solutions.

Stop seeking an easy way; look for ways to facilitate the hard work.

Distinguish between problems caused by technology and fundamental ones.

- If its a technological problem, remove it at source.
- If it is a fundamental problem, find supporting technology

Documentation is a real, and fundamental, problem that will not go away.

We can produce documentation that is much more useful than what we have.

There are *real* mathematical research problems in documentation tools.

We can produce incredibly useful documentation based tools.

Moving ahead means working on old problems, fully aware of old approaches, and doing something better.

SQRL wants collaborators. There are problems for everyone.

• **SOFTWARE QUALITY RESEARCH LABORATORY** •

