University of Limerick

Document Driven Testing and Inspection

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)[1]
Faculty of Informatics and Electronics
University of Limerick
Republic of Ireland

Abstract

Document Driven Disciplined Design can improve the quality of software used in designing products, manufacturing products, and as components of the products themselves. This lecture discusses documentation based quality assurance procedures.

[1] SQRL is funded by Science Foundation Ireland (SFI) and associated with the Irish Software Engineering Research Consortium (ISERC).

What Do I Mean By "Good" Software?

Good software does more than "work correctly most of the time".

- The users must know that it can be trusted. (vertrauenswürdig)
- Unavoidable failures should be "safe".
- When something goes wrong, it must be easy to find "guilty" module; corrections should be confined to that module and not affect the rest.
- We must be able to revise software quickly and safely when needs change.
- It must have subsets with which we can start to do useful things.
- It should be extensible - without altering previous subsets.
- It must have firm, precisely defined internal and external interfaces.
- It must be able to survive the departure of its developers.
- It must "age gracefully" as it is updated, corrected and improved.

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 (Nuclear plant):

- 218 discrepancies found after 6 years of careful testing.
- 15 years of active use and change with only one error (analysis) found

When is Software a Critical product?

"Critical" does not necessarily mean "safety critical"

Other types of critical programs:

- Mass distributed programs in warranty situations
- Critical kernels in many systems
- Financial Systems
- Security (Privacy, Data Protection) programs
- Any system where a failure may lead to a lawsuit
- Administrative systems for public organizations

The common property of all of these examples is that the cost of a failure is high.

If you value your reputation, your work may be critical.

The Critical-Software Tripod

(1) Precise, well organised, mathematical documentation with systematic review

(2) Extensive Testing

- Systematic Testing-quick discovery of gross errors
- Random Testing -discovery of shared oversights and reliability assessment

(3) Qualified People and Approved Processes

These three legs are complementary

- The three legs are *all* needed.
- The tripod will not stand if any leg is forgotten.
- The third leg is the shortest - it limits our ability to extend the others.
- Today we discuss only leg (1).

Twelve Reasons Conventional Reviews are Ineffective

(1) The reviewers are swamped with information.

(2) Most reviewers are not familiar with the product design goals.

(3) There are no clear individual responsibilities.

(4) Reviewers can avoid potential embarrassment by saying nothing.

(5) The review is a large meeting; detailed discussions are difficult.

(6) Presence of managers silences criticism.

(7) Presence of uninformed reviewers may turn the review into a tutorial.

(8) Specialists are asked general questions.

(9) Generalists are expected to know specifics.

(10) The review procedure reviews code without respect to structure.

(11) Unstated assumptions are not questioned.

(12) Inadequate time is allowed.

Active Reviews are Effective Reviews

A dilemma:

- Errors should be found *before* the documents/systems are used.
- Errors are usually found *when* the documents are used.

Another dilemma:

- Everyone's work requires review!
- It's easier to say "OK" than to look for and report subtle errors!
- Reviewer's approval is not reviewed.

One more dilemma:

- No individual can review all aspects of a design.
- In a group, people often relax knowing that others are working on the problem.

Solutions:

- Make the reviewers use the documents and document their analysis.
- Have specialised reviews. Ask the reviewer about things they know.
- Make the reviewers provide specifics - not just a ok/not ok.

Previous Work on Inspections

Best known publication Fagan - 1976. Many followers - new book by Gilb.

Explicitly focus on the **management** issues.

- Who should be there?
- What are the roles of the participants?
- How long is a meeting?
- How fast do you work?
- Forms for reporting errors?

Read the code in sequence and paraphrase it in a natural language.

Paraphrases are informal.

Focus on estimating number of errors, not eliminating all errors (Gilb).

Most observers find these more effective than conventional reviews or walk-throughs, but... can we do better?

Parnas/NRL/AECB/AECL/Ontario Hydro

Focus is more technical.

Depends on hierarchical decomposition rather than sequential reading.

Uses mathematical notations to provide precise descriptions rather than informal paraphrases.

Produces useful *precise* documentation as a side effect.

- Proceed much more quickly if the documentation has already been produced by the developers.
- Documentation also useful in the testing process.

Insures that cases and variables are not overlooked.

Applies mathematics to check for completeness and consistency of documents.

Can take advantage of mathematics based tools such as theorem provers and computer algebra systems.

Active Review of Design Documents

Base the review process on the nature of the document.

(1) Begin by identifying and listing desired properties.

(2) Prepare questionnaires for the reviewers. Ask them questions that:

- make them use the document.
- make them demonstrate that the desired properties are present.
- ask for sources of information to support the answers to other questions.

For example:

- Ask reviewers to identify the domain of the program
- Ask reviewers to identify “error” cases.
- Ask reviewers to explain why no other error cases are possible.
- Ask reviewers to explain why the behaviour required for each case is the desired behaviour.

Beyond Inspecting Documents - Inspecting Programs

It is the code that "hits the road".

Getting the requirements right, the structure right, the interfaces right, the documentation right, etc. are all important but *we have to check the code*.

The same review principles apply, viz:

- Make the reviewers use the material they review.
- Make the reviewers answer questions.
- Ask the reviewer about things that they know.
- Make the reviewers provide specifics.

We compare completed programs with previously reviewed specifications.

We ask some reviewers to produce precise descriptions.

We ask other reviewers to show that the descriptions match the specifications.

It is hard work but it produces results.

- We get good documentation for future use.
- We find errors in thoroughly tested programs (considered correct).

SQRL Code Inspection Process

(1) Prepare a precise specification of what code should do.

(2) Decompose the program hierarchically into parts.

(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 have been 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 bunch of small steps and very systematic.

Descriptions vs. Specifications

Definition:An actual description is a statement of some actual attributes of a product, or set of products.

Definition:A specification is a statement of all properties required of a product, or a set of products.

In the sequel, “description”, without modifier, means “actual description”.

The following are implications of these definitions:

- A description may include attributes that are not required.
- A specification may include attributes that a (faulty) product does not possess.
- The statement that a product satisfies a given specification is a description.

The third fact results in much confusion. A useful distinction has been lost.

Hierarchical control structure in programs

"Structured" Programming constructs have three very useful properties:

(1) programs constructed using them can be decomposed into a hierarchy of parts (with lower level parts completely contained in an upper level part) using simple parsers; those parsers need not even distinguish one identifier from another,

(2) the semantics of the total program can be determined from the semantics of its parts[1], using simple operations (cf. e.g. [14, 15]).

(3) semantics can be determined in a simple order: inner parts first.

The above properties make it easier to study a long program.

The Display Method takes advantage of those properties and is intended to be used for programs that have these properties.

---

[1] It is an unfortunate property of today's programming languages that the semantics of the components depends on things out side those components.

Use of Data Abstractions

The best structured program will be difficult to explain and understand if it is presented in terms of complex data structures.

Data structures should be hidden by the introduction of *abstract data types*

Precise program documentation is not possible unless the abstract data type interfaces are precisely documented.

The following examples have been selected so that they can be understood without an understanding of module specifications.

Documenting Longer Programs Using Displays

With a good architecture, programs will be shorter.
There are still long programs.
Long programs can only be understood if you understand their structure.
We can document them in a way that makes their structure explicit.
This should be done by the developer - not those who come afterward.
It can be done afterward.

Documenting Longer Programs Using Displays

A display consists of 3 sections:

- A small program identified in the hierarchical decomposition
- A specification stating what that program must do.
- Specifications for the programs used by that program.

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

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

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

- standard programs - defined in standards documents.
- primitive programs - defined in company dictionaries

Manually check each display. Completeness check is automated.

University of Limerick

Decomposition

(**integer** array H[1:N];
(**integer** c; **integer** n; n ⇐ 1;
it (n ≤ N →
(
(**integer** u; **integer** *l*; **boolean** p; *l* ⇐ 1; c ⇐ 0;
it (u ⇐ *l* + n -1;
(u ≤ N → (

(**integer** i; i ⇐ 0; p ⇐ **true**;
it (i < ⌊(u - *l* +1)÷2⌋ →
(A[*l*+i] = A[u-i] → (i ⇐ i + 1; ☞)
| A[*l*+i] ≠ A[u-i] → (p ⇐ **false**; ●))
| ⌊(u - *l* +1)÷2⌋ ≤ i→ ●)
ti)
;
(¬p → **skip** | p → c ⇐ c+1); *l* ⇐ *l*+1; ☞)
| u > N → ●))
ti)
;
H[n] ⇐ c; n ⇐ n +1; ☞)
| n > N → ●)
ti))

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 statistically significant.

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.

Novice Approaches To Testing

(1) Eureka! It ran.

- one successful test or a few successful tests means its done

(2) A number of tests where the answers are easily checked

(3) Let it run and run and run.

(4) If an error is noticed, fix and go to 2.

(5) Test until the product is due.

What's wrong with this?

- Much of the program may never be tested.
- We get is a bunch of anecdotes, but do we know anything beyond those cases.

The hardest question, "Have we tested enough?".

- How important is the product?
- How costly would it be to release an incorrect product?

Some Better Approaches to Testing

1) Test every statement at least once.
2) Test every exit from a branch at least once.
3) Test all possible paths through the program at least once.

These are minimal requirements, but consider a program containing:

$$y := sqrt(|x|)$$

and another program in which the above is replaced by:

if x < 0 then y := sqrt(-x);
if x > 0 then y := sqrt(x);
if x = 0 then y := 0;

Does the first program need less testing than the second

The above rules assume that program state is more important than data state.

In fact, program state and data state are distinguished only in our minds. We often trade program complexity for data complexity.

Additional Rules

(4) Consider all typical data states
(5) Consider all degenerate data states
(6) Consider extreme cases
(7) Consider erroneous cases.
(8) Try very large numbers
(9) Try very small numbers
(10) Try numbers that are close to each other.
(11) Think of the cases that nobody thinks of.

Nobody can do all of the above.

Nobody can be sure that they have done all of the above.

Who Does the Testing?

You are a fool if *you* don't test your program!

- If you do not know some fact, no amount of inspecting your program reveals it.

Your *customers/bosses* are fools if *they* don't test your program?

- If you overlooked a case in programming, you are likely to overlook it when testing.

Many companies have testing specialists in quality assurance departments.

"Cleanroom" model says that you are not allowed to test your program.

- Increased care yields big improvements in quality.
- Statistical testing is done by others.

If you have independent testers, they require specifications. Without them the testing leads to a costly debate about the specification.

Random vs. Planned Testing

People tend to overlook the same cases. How can we get around this?

Can *random* testing be better than *planned* testing?

Can we plan random testing?

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.

Even in "Black Box" Testing, What's Inside Makes a Difference!

The number of tests needed to identify a finite state machine depends on having an upper bound for the number of states.

- Unless you have an upper bound, you can always invent a machine that passes **all** previous tests but will fail another one.
- Any statements that we make about thoroughness of testing must be based on knowledge of this upper bound.
- The "tighter" the upper bound, the better your testing strategy.

This is particularly vital when considering interactive systems with memory.

- What is the length of a test sequence?
- Why are 200 tests not considered to be a single long test?
- A test sequence is "over" when you return to a previously tested state, but how do you known when that has happened.
- A pure black box strategy cannot be sound in those situations.

The length of test-trajectories may be limited by re-initialisation.

Another Testing Method Classification

Planned Testing

- Clear Box - based on data state and code coverage criteria
- Black Box - based on external case coverage and interface specification

"Wild" Random Testing (tests picked without use of knowledge about usage or specification beyond syntactic interface).

- Pick arguments using uniform random distribution.
- Can violate assumptions yielding spurious errors.
- Any reliability figures obtained aren't meaningful.

Statistical Random Testing.

- Requires knowledge of expected usage patterns (operational profile)
- Testing is designed to approximate actual usage
- Provides meaningful reliability estimates
- Only as good as the operational profile.

What are the Limits of Software Testing?

"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.

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!

What Does it Mean to talk about Software Reliability?

Software is not a random process.

Software doesn't usually fail randomly; its behaviour is predictable

It is the input data that introduce randomness.

"Software Reliability" is a measure of the input distribution through a boolean filter.

Software + Hardware should be assessed as a single component. It is generally difficult to filter out the hardware effects.

Two Meaningless Measures of Software Reliability

- The number of errors per 1000 lines.
- Time derivatives of the number of errors per line.

We never know these numbers - we know only what we found.

They are counting the number of corrections made, not the number of errors.

- One error may require correcting many lines.
- Sometimes many lines are corrected when only one was wrong.

The “experts” are measuring the reliability of their programmers, not the reliability of their programs.

For user and maintainer, it is the failure rate that matters.

How much testing is needed to assess reliability?

(1) Assume that we have the right input distribution (difficult). We will use tests selected randomly from this distribution.

(2) Let 1/h be the required reliability.

(3) What is the probability of passing N properly selected tests if each test would fail with probability 1/h?

$$M = (1 - 1/h)^N$$

(4) M is the probability that a marginal product would pass a test of length N.

Some examples for h = 1000

- N = 500, M = 0.606
- N = 1000, M = .36700
- N = 5000, M = .00672

Some examples for h = 1,000,000

- N = 1,000, 000, M = .36788
- N = 5000000, N = .00674

Systems with high reliability requirements, are much harder to test. Some companies use customers to get more tests.

Real-time systems are harder to test than batch (memory free) systems

In real-time systems, a test is a trajectory, not an input state.

The trajectory must be long enough that sleeper bugs are revealed.

There must be an upper limit to the memory of the system.

Systems must be structured with testing in mind.

Most of the memory must be periodically reinitialised.

Testing must be repeated for each mode of the remaining memory.

Defining the probability distribution of trajectories is the hardest part.

Inspection is more important in real-time applications than batch programs.

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.

Summary

It's what's up front that counts. The time required for testing and inspection is greatly reduced if the required documents were produced during design.

Testing and Inspection are complementary and both can be enhanced by precise documentation.

For inspection, it is "divide and conquer" that makes the difference.

- Precisely documented hierarchical, information hiding, architecture.
- Hierarchically structured programs documented as displays
- Tabular expressions
- Being systematic all the way through.

For testing, the key is precise specifications.