Testing is sometimes viewed as an “add on” step in software development - something you do to demonstrate that the product is ready for use. Test planning is often postponed until the development is near its end. This results in incomplete testing, ambiguous test results, and the release of products of doubtful quality.

After reviewing fundamental software testing issues, we describe a document-driven testing approach in which test plans and test result evaluation are done with the aid of documentation prepared throughout the design process. The policies about testing, and response to test results are determined in advance and high quality standards can be enforced on a project.

These slides, and others from the SQRL lecture series, are available at:

http://www.csis.ul.ie/Research/ParnasLectures/Default.HTM
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. 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:

\[
\begin{align*}
\text{if } x < 0 \text{ then } y &:= \sqrt{-x}; \\
\text{if } x > 0 \text{ then } y &:= \sqrt{x}; \\
\text{if } x = 0 \text{ then } y &:= 0;
\end{align*}
\]

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.
Hierarchical Testing Policies

Testing the whole system at once is usually a mistake.

- Finding the fault is a nightmare.
- May be many simultaneous faults.

Test Small Units first.
Integrate after components have passed all tests.

Test lower levels of uses hierarchy before using them.

Bottom Line:
Testing before integration will save time after integration.
Component boundaries must be precisely defined or the units cannot be tested independently.

• SOFTWARE QUALITY RESEARCH LABORATORY •
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.

More basic issues:

- All 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?

*SOFTWARE QUALITY RESEARCH LABORATORY*
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.
All Three Types Of Testing Have Their Place

Black Box tests:

- can be re-used with new design
- can be developed independent of designer - good for acceptance testing.
- test what you really care about.

Clear Box testing:

- takes advantage of our knowledge about the program and basic programming errors.
- provides an objective measure of coverage.
- tests programmer’s “work product”.

Grey Box testing:

- gives better coverage for black box with memory.
- can avoid some duplicate tests by detecting return to same state.
- provides a basis for a measure of testing thoroughness.
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.
Three ways to Measure Software Quality

All measures assume there is a specification distinguish right from wrong.

Correctness:

• Does the software always meet the specification?
• Correct software always meets its specification.

Reliability:

• What is the probability that the software will work as specified?
• Can be on a per run or per time period basis.
• Answer depends on usage.

Trustworthiness:

• What is the probability that catastrophic flaws remain?
• Of interest when even a single failure is unacceptable.

Each measure requires a different approach to quality assurance.
When Should we use each of these Quality Measures?

Correctness:

- Rarely need it!
- Nice to reach for, hard to get.
- To a perfectionist, all things are equally important; realists take a different attitude.
- Not our real concern, we accept imperfections.
- If you have a small finite state space, you can do an exhaustive trial.
- Binary Decision Diagrams, (BDDs) handle slightly bigger cases.
- In other cases, use mathematical methods and rigorous proof.
When Should we use each of these Quality Measures?

Reliability:
- Most relevant when we can consider all errors are equally important,
- Useful when there are no unacceptable failures.
- Meaningful when statistical characteristics of input are predictable,
- Meaningful when we can talk about the expected cost, (e.g. warranty).
- Useful when your main concern is inconvenience.
- Useful when we want to compare products.

To estimate reliability use statistical testing.
To measure reliability retain data from actual usage.
When Should we use Each of these Quality Measures?

**Trustworthiness:**

- Use when you can identify unacceptable failures,
- Use when trust is vital to meeting the requirements,
- Use when there may be antagonistic “users” (statistics mean little in that case).

We often accept the systems that are unreliable.

We do not use systems that we dare not trust.

Testing alone does not work for trustworthiness unless the state space is very small.

Testing can be combined with proof by using proof to reduce the state space.

To achieve trustworthiness use formal documentation and systematic inspections.
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!
Practical Limits of Software Testing?

(1) It is not usually practical to prove correctness by testing.
(2) Testing cannot predict availability without data on repair time.
(3) Reliability predictions based on old versions are not valid.
   • This is a controversial statement. Some do not want to discard the data.
   • You would have to assume that the failure rate is not affected by the repair.
(4) Testers make the same assumptions as the programmers.
(5) Planned testing is a source of anecdotes, not data (H.D. Mills).
(6) Built-in self-tests test for decay, but not designed-in defects.
(7) Generally, a lot of testing is needed.
(8) Generating the test cases requires a lot of work.
(9) Evaluating the results requires at least as much work.
(10) Specifications are needed but not widely available.
   • Results are often ambiguous - lead to debates.
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 $h = 1000$

- $N = 500$, $M = 0.606$
- $N = 1000$, $M = 0.36700$
- $N = 5000$, $M = 0.00672$

Some examples for $h = 1,000,000$

- $N = 1,000,000$, $M = 0.36788$
- $N = 5000000$, $N = 0.00674$

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?

Let’s review some of the problems about testing.

- 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.

We will discuss each of these in turn.
Component Testing Requires Defined Component Boundaries

When doing “unit testing” you will not be seeing how well two components interact - you are testing each component separately.

It is essential that you have a specification for that component that is also used by the programmers who are implementing it and those who are writing programs that use it.

The module specifications described earlier are perfect as a specification.

• They can be checked for completeness and consistency.
• Simulation can be used to check if you have specified what you want.
Statistical Data is Needed for Reliability Estimation of Components.

Operational profiles can be expressed in terms of “traces”

- For each canonical trace, specify the likelihood of an extension
- Distributions for parameter values can also be specified.
- The equivalence classes used need not be the same as in the specification.
- Traces can be generated with this statistical data and used for testing.
- Denise Woit has done extensive research on this approach.
Some Components may be Ready for Testing Before Others.

Modules not yet implemented can be simulated on the basis of these specification.

The implementors can use the same simulation to test their own work.
Generating Test Cases is Prone to Oversights, Very Time Consuming

Test case generation can be automated on the basis of a module specification.

- Statistical generators can be used as discussed earlier
- Test cases can be generated by solving equations (von Mohrenschildt)
- Test coverage can be measured and displayed in tabular form.
Evaluating Test Results is Very Time Consuming and Prone to Oversight.

Test oracles can be generated from the specification.

- Program function tables can be translated into boolean expressions provided that all quantifiers are over finite, generatable, sets. ““Dennis Peters)

- Module specifications can be translated into simulations using term rewriting techniques (Yabo Wang)
  - Note that simulators are more than oracles.

- Requirements documents can be translated into run-time monitors. (Dennis Peters)
  - Note that these only sample - continuous monitoring is not possible.
  - Monitor can also be used as a safety system during operation.
  - Errors in requirements document may occur as well errors in code.

- Module design documents can be used in grey box testing.
  - One can generate an oracle from the abstraction function.
Difficult to Generate Test Cases Until Program is Complete.

In industry, testers often do not have enough information to test until they are given the program to test.

Specifications are open to interpretation.

Specifications are subject to change.

We can do something about the first two problems by writing a module spec. Test cases can be generated automatically from the specification or by hand. We cannot do anything about the third problem except try to discourage it.

If we have automated test case generation, changes in specifications are easier to handle.
Difficult to Evaluate Thoroughness of Test Coverage.

A table with the same structure as a program function table can be generated from test data that is generated some other way.

This will display the coverage. (Nan Nan Wang)

Coverage of canonical trace classes can also be measured and displayed.

Coverage of abstraction function sub domains can also be measured and displayed.

Reliability estimates and their confidence level can be computed.
University of Limerick

Conclusions About Testing

1) Time spent in documentation can be “paid back” in testing.

2) Testing is difficult, but good documentation will make it easier, much easier.

3) Engineers see testing and mathematical analysis as complementary and both are necessary.
Conclusions About Developing High-Quality Software

Good software can be achieved only by disciplined hard work. That work includes careful attention to:

• precise specification of the requirements that the software must satisfy
• decomposition of the software into components
• design and documentation of module interfaces
• precise documentation of internal module design decisions
• writing well structured programs

followed by:

• systematic inspection of all code and documents, and
• disciplined testing

Precise Documentation is the key.
Support tools are essential.

• SOFTWARE QUALITY RESEARCH LABORATORY •
The Software Quality Tripod

1) Precise, Organised, Mathematical Documentation and Extensive 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.
University of Limerick

**SQRL:**

Building a team of researchers to develop, methods, tools, examples.

- Some will develop tools
- Some will develop methods
- Some will work with industrial developers to train and evaluate.

If you know a person who can join the SQRL team, tell them to contact me.
We are looking for a few “nuts” who want to be different and make a difference