Software Quality Research: Why, What and How

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)
Department of Computer Science and Information Systems
Faculty of Informatics and Electronics
University of Limerick

Abstract

UL is establishing a Software Quality Research Laboratory (SQRL). This lecture will discuss why research on ways to improve software quality is important, what topics will be studied by SQRL at the University of Limerick and how the research will be conducted. Intended for the broader University and Industry community, the lecture assumes no expertise in software development. It will be followed by six more technical lectures for software developers and managers; those weekly seminars will discuss:

- Precise specification of software requirements
- Decomposition of the software into components (modules)
- Design of module interfaces
- Writing hierarchically structured programs
- Precise documentation of design decisions
- Systematic inspection and disciplined testing

Outline

1. What are *software* and *software engineering*: a variety of views
2. The growing importance of software
3. The gap between theory and practice in software development
4. An engineering educator's view of programming education
5. Building a true profession - debate about licensing Software Engineers.
6. Six essential activities in software development
7. SQRL research
8. Five years from now

Software Engineering - a multi-faceted field

The five blind men and the elephant

Software development produces a lot of “blind men”. They see:

- a cost estimation problem
- an adaptability problem
- a scheduling problem
- a human interaction problem
- an interface specification problem
- a programming language problem
- a version management problem
- a mathematical verification/validation problem
- a performance prediction problem
- a software design methods problem, ...
-

My own eyesight is too weak to make a complete list.

Even the “Union View” is not the whole picture

All of these views focus on the inside of the computer.

Software development requires “external” knowledge.

- The environment: the physics etc.
- Mathematics
- Operations research
- Business models

The view of Software Engineering taken by Computer Science Departments is generally very narrow.

If Software Engineering is viewed as a branch of Engineering, it is not just a name or jurisdictional change, the content is different.

Why Today's "Software Engineers" are not "Engineers"

Consider the task of writing software to control the temperature of a house that has a hot-water/radiator heating system.

To a CS graduate this problem looks simple. Simulate a person or a thermostat with a little bit of software that senses temperature and switches the heating on and off if the room is too warm or too cold.

In fact, this might not be inadequate. It could allow excessive temperature swings.

We cannot even evaluate the adequacy without a model of the house and heating system behaviour.

Many "software engineering" educators have questioned the need for basic science (physics, chemistry) in software engineering" programmes.

There is a real need for programmes in "Software Intensive Engineering" as well as professional programmes in "Software Development".

What is software?

This is not a simple question.

- When is a program software? Are all programs software?
- Is data software? Can one really distinguish between programs and data?
- Is this a legal issue or a technical one?

There are many views.

- multi-version, multi- person programs
- initial state of a machine
- a way to build “virtual” machines that are more powerful (more convenient)
- a way to build “virtual” machines that are less powerful (more restricted)
- tailoring device turning a general purpose tool into a special purpose tool
- What we produce with our tool is software, nothing else is software.

This is one of those questions that is not worth asking; you’ll know it when you see it.

Each View is Important

Multi-version, multi-person programs

- This view determines what a software designer must know beyond programming.

Initial state of a machine

- Explains what happens when we start or install software
- Helps to understand the program/data trade-off decisions
- Takes anthropomorphic analogies out of our analysis

Building “virtual” machines that are more powerful (more convenient)

- This is what allows us to sell software.

A way to build “virtual” machines that are less powerful (more restricted)

- This is how we should build safer, more secure, easier to use, software.

Tailoring device: turning a general purpose tool into a special purpose tool

- Hardware is mass produced, good for everything, good for nothing; Software makes it good for something.
- Explains why there will always be more jobs in software development than in hardware design.

The Growing Importance of Software

Three periods in my working life:

- Software out of sight in large, relatively rare, mainframe computers
- Software visible in the computer on everyone’s desk
- **Invisible software** (contained in devices or used to design devices)

Ubiquitous and critical

- 1969: Safety requirements for software was a joke
- 1999: People die when software fails

Software is unreliable and untrustworthy

- Who has not experienced many software failures?
- People die when software, or a software designed product, fails.
- Compare a new car with a new software product.

All industry will be helped by better software

- The “software industry” may be helped least of all.

Who Depends on Software Development and Acquisition?

It is an error to think that only software companies have software problems.

Car companies, oil companies, shipping companies, insurance companies, drug companies, power generation companies, banks, ... all have problems with software development and acquisition.

Software companies have their own experts and think that they know the answers.

Smaller companies, and companies that do much more than software, know that they need help.

My conclusions are clear:

- Almost every sector in Ireland would be strengthened by increased expertise in software, organized in an effective way.
- This is not the best time to cater to the “hi-tech” companies.
- This is the time to build capabilities to make excellent software products by research and education.

The Gap Between Theory and Practice in Software Development

Four broad areas of Computer Science:

- Theory (automata, formal methods, algorithms, complexity, computability)
- Programming tools (compilers, operating systems)
- Program development methods (Jackson, Yourdon, ..RUP, XP...)
- Design of specific software products

These divisions keep growing and the areas split, split, and split again.

The isolation of these fields seems unprecedented in Engineering.

Academia wears rose coloured glasses, shuts its eyes to reality.

- Program verification is viewed as a solved problem.
- “Information Hiding” isn’t enough.

Industry accepts the world as it is, shuts its eyes to academia.

- Won’t read research papers (unlike traditional engineering)
- Most of the time they are right. Research papers are not written for them.

Computer Scientists talk to “Computer Engineers”

Computer Scientist: “Interesting consulting -where’s the research?”

Engineer: Applications? “What are the applications?”

Computer Scientist: “Application -- simplifying proofs in number theory.”

Engineers: “Which language should we teach?”

Engineers: “What does a termination argument have to do with programming?”

My conclusions:

- Most Engineers, even those who write software, know very little CS.
- Most CS researchers have no idea what would constitute useful Engineering research.
- The gap between theory (research) and practice keeps growing.

An Engineering Educator's View of Programming Education

My first shocks(1962):

- Arbitrary human decisions vs. fundamental laws of nature
- Show them nice designs instead of teaching how to design.
- Teach how to prove a theorem rather than how to use a theorem.

The same things happen today.

We all tend to produce younger versions of our self (or the person that we wish we were).

- Scientists tend to prepare more scientists.
- Mathematicians tend to produce young mathematicians.
- Engineering educators should prepare young people for product development.

Most of our computing students end up as product developers.

Teaching Science vs. Teaching Engineering

Both Engineering students and Science students need to learn Science.

It is the understanding of fundamental science and mathematics that distinguishes the educated Engineer from a trained technician.

Technology changes rapidly; our knowledge of science advances slowly.

Scientists and Engineers use their science knowledge differently.

- Scientists must know facts and how to find more (or more accurate) facts
- Engineer must know facts and how to use that knowledge in product development.
- Engineer’s knowledge must be broad; Scientist’s knowledge must be deep.

Our educational programmes must reflect that difference.

All of our science and engineering students must understand the difference between science and today’s technology.

Their knowledge of science must help them to keep up with technology.

What is the “Science” for Software Developers?

Engineering depends on knowing reality, but

- What constitutes *nature* in the software world?
- *Nature* is not ‘what developers naturally do’.
- *Nature* includes what developers can or cannot do.
- *Nature* is not a set of arbitrary decisions by those who came before us.

We must distinguish what can be changed from what cannot be changed.

- This is what distinguishes a realist from a pessimist. Microsoft’s interfaces are not fundamental laws.
- Most of the fundamental science of Software Engineering is mathematics.

Teaching fundamental theory is important but not enough.

- We have to teach Engineering students how to use that theory in design.
- We have to teach Engineering students how to use that theory to understand new technology.
- ‘If we cannot teach them how to use it, or how it affects practice, perhaps we should not be teaching it *to Engineering students*.

Lack of Demand and Reduction in Student Interest

Nothing grows forever and the IT field is no exception.

Many consumers already have for more communication and processing capacity than they will ever need.

Where once “any warm body” could find a position, today there are good students looking for jobs.

Student interest has reduced accordingly. Is it a concern?

Many of our students were neither suited for, nor really interested in, the fields they were studying.

We should focus on educating graduates who want to be really good software developers for the next 40 years.

- teach them the fundamental principles
- teach them how to use those principles today and in the future.
- Teach only what would have been useful 20 years ago.
- Teach only what we think will be useful 40 years from now.

Should Software Engineers be Licensed?

You need a license to cut people’s hair, install a gas fitting, etc.

You do not need certification to design, modify or install critical software.

In the U.S. and Canada you must be licensed to practice Engineering?

For regulated fields, a core body of knowledge is enforced.

Why should Software Engineering be different?

- Rapid technological change is **not** a difference.
- Practitioners, researchers and educators come from unregulated fields.
- Researchers ego interferes with analysis.
- The core body of knowledge has not been identified.
- Sub disciplines have not been identified.
- Software is invisible.
- There are few who would be qualified.

Political Issue: Those who are endangered should be more active.

Professional Issue: The core body of knowledge must be specified.

Six Key Activities in Software Development

Developing High Quality Software requires careful attention to:

- Precise specification of software requirements
- Decomposition of the software into components (modules)
- Design of module interfaces
- Writing hierarchically structured programs
- Precise documentation of design decisions
- Systematic inspection and disciplined testing

Precise Specification of Software Requirements

You are unlikely to produce the right thing by accident.

Decisions about function should not be made by programmers while programming.

Finding all the requirements requires a systematic method.

Document must be designed as a reference document

- Strict rules of content organization.
- Precisely defined terminology and notation.

Most software development organizations do not do this, do not know how to do this.

You never have time to do it up front, but you always have time to fix it at the end. (Fred Brooks, B.O. Evans)

Long Term Savings (Boehm)

Even the best of today’s documents leave too many decisions to the programmer and postpone user-review to the end of the development.

Decomposition of the software into components (modules)

The smallness of the human skull (E.W. Dijkstra)

The “Mythical Man Month” (Fred Brooks)

Inevitability of change

Cost of “ripple effect” during change

Reducing complexity by “divide and conquer”

Can be done incredibly badly

Can be done very well - improving quality in many ways

Design decision - not just a managerial decision

Today:

- Most developers do not give it serious thought!
- Most developers do it wrong (follow processing steps).
- Improvements are possible even in existing software.

Design of Module Interfaces

The “flip side” of decomposition:

Communication between components is essential.

Interfaces should be “as simple as possible but not simpler”.(A. Einstein)

Incomplete interfaces lead to many annoying “glitches”.

Interfaces should be designed not to change.

- Serious analysis of change possibilities is necessary.
- Understanding how to design abstractions is essential.

Today:

- Most software developers do not take the time.
- Change is the rule, not the exception, in component interfaces.
- Complex interfaces are difficult to document.
- Complex interfaces make the modules more complex.

Writing Hierarchically Structured Programs

Every module is a set of programs.

Divide and Conquer is the key to getting programs right.

Strictly hierarchically structured programs are easier to divide.

The functions of the subprograms are easier to summarise.

Each level is easier to analyse.

Hierarchical structure is always possible (Boem-Jacobini).

Finding a *good* hierarchical structure is a creative and difficult process.

Taking advantage of that structure requires careful analysis.

This aspect of programming is language independent.

Today:

- Most practitioners do not bother to do it.
- If it is done, subsequent changes destroy the structure.

Precise Documentation of Design Decisions

In many development organizations there is no shortage of documentation.

- It is untrustworthy, incomplete, inconsistent, out of date
- Only its author can use it.
- Time to update or correct documentation is considered wasted.
- XP and similar movements advise against it.
- Words words words and more words
- There are no content organization rules.

Today:

- Time is wasted trying to get information.
- Mistakes are caused by inaccurate or inconsistent information.
- Changes introduce new errors because of poor documentation.
- Review is difficult because of lack of documentation.
- Documentationis often relegated to people who cannot program.
- Documentation is often done after the fact by people who do not know the facts.

Engineers Design Through Documentation

A series of increasingly detailed documents record design decisions.

Documents are used for review.

Documents serve as the basis for later design decisions.

Documents are strictly organized; notation is well defined.

Documentation is essential during maintenance, revision.

Engineering documents use mathematics explicitly and implicitly.

Why should Software Engineering be so different?

- Nobody knows how to do it
- No Engineering traditions
- Short-sightedness
- Many unqualified practitioners

Because of its complexity, documentation is more important for software than for other types of products.

Systematic Inspection and Disciplined Testing

To err is human and program developers are very human.

Complex products are very difficult to inspect.

Software products are unusually difficult to test.

- Product state spaces are very large
- Environment state spaces are very large
- Complete testing is not feasible
- Test case selection is difficult, depends on purpose
- Statistical reliability estimation requires a lot of knowledge.

Testing and inspection are complementary.

Systematic documentation supports both.

Discipline can be based on the documentation.

Today:

- Most practitioners do not know these methods.

SQRL Research Goals

The picture that I have painted is too bleak.

- The technology available to us is incredibly powerful.
- We already do some amazing things.
- The industry is full of enthusiastic and creative people.

But, we do not use that potential well- we can do better.

Software can be more reliable, more trustworthy, easier to use, and cheaper.

The key problems are education and qualification standards.

Those problems are exacerbated by the lack of material to teach.

Research is needed to develop that “core body of knowledge”.

SQRL will do research that will discover, publish, and transfer:

- methods
- notation
- support tools.

SQRL’s Starting Point - I

The starting points for SQRL’s research are the following observations:

- Engineers use documentation as a design medium.
- Engineering documentation is based on mathematics and science.
- The “science” of software is primarily mathematical knowledge.
- Engineers’ view of mathematics is different from that of pure mathematicians and closer to that of so-called applied mathematicians.

Consequently,

- Documentation is central to our approach.
- Mathematics will play a big role in our work.

We have rejected popular (among academics) “formal methods” because:

- They are not “practitioner-friendly”.
- They are based on a pure mathematics view of mathematics.
- They provide models rather than specifications.
- They do not summarise (scale up) well.

SQRL’s Starting Point - II

Previous research results:

- Information-hiding approach to decomposition (prehistoric)
- Semi-formal Documents for some real systems (early modern)
- Relational Models of Document Contents (modern)
- Tabular Expressions (ongoing)
- Exploratory tools (all possible because of the mathematical model)
    - tabular expression storage kernel
    - expression input tools (several)
    - expression evaluation
    - test oracle generator, monitor generator
    - test case generator
    - inspection management
    - completeness checker
- A variety of case studies - every one a learning experience that advanced the associated project.

Two Complementary SQRL Advantages

Relational models of document contents

- precise definition of required document contents, not just formats
- “black box” descriptions, not implementations or models
- useless if you do not have a specified notation and format

Tabular Expressions

- Are as precisely defined as any other mathematical expressions
- Expressions contain conventional notation in each cell.
- Various formats can be chosen to fit nature of the function.
- No axioms: Entries are either closed form expressions, or predicates such as equations and inequations.
- New, but not foreign to Engineers

A Simple Conventional Expression

$$(((\exists\ i, \mathrm{B}[i] = x) \wedge (\mathrm{B}[j'] = x) \wedge (present' = \mathbf{true})) \vee ((\forall\ i, ((1 \le \mathrm{i} \le \mathrm{N}) \Rightarrow \mathrm{B}[i] \ne x)) \wedge (present' = \mathbf{false}))) \wedge (\text{‘}x = x\text{’} \wedge \text{‘}B = B\text{’})$$

An equivalent tabular expression:

	$(\exists\ i, \mathrm{B}[i] = x)$	$(\forall\ i, ((1 \le \mathrm{i} \le \mathrm{N}) \Rightarrow \mathrm{B}[i] \ne x))$
j’ |	B[j’] = x	*true*
present’=	true	false

$\wedge$ NC(x, B)

The above is one of many kinds of tables!

Simple tables like this *understate* the advantage.

These have “practitioner appeal”.

- Can be used with any mathematical method.
- Can be used without quantifiers, but finite quantification is analogous to summation.
- Informal tables are useful and provide a stepping stone to precise documents.

Tables Don’t Make a Difference: Theoretically

We cannot express anything with table that we could not express without it.

The proof rules don’t change because the underlying abstraction is the same.

Why do we advocae them?

- They are easier to write, people make fewer mistakes.
- They are easier to read.
- They can be used as reference documentation.
- They help in a “divide and conquer” approach to design and inspection.
- They encourage and support systematic disciplined thinking.

When we ask students to write descriptions in conventional notation, they get them wrong. With tables they usually get them right. The same thing has been noted in industry.

Tables ***really*** make a difference but systematic research must replace anecdotal evidence and will lead to even more improvement.

Is That All There Is?

No!

With these concepts we can do more than ever before.

With these concepts people make fewer mistakes.

With these concepts we can use mathematics based tools, but

We are not yet ready to roll.

- With complex data structures, expressions are still hard to read.
- Developers need more preparation in discrete mathematics.
- All of our tools are pre-prototype
- The set of tables is still too restricted.

However, the first “round” of work on these methods has had enough success to justify a larger investment.

Software Development Methods Must be More than Sound

Sound models can be purely academic research, but making a difference cannot be purely academic.

Some, perfectly sound, methods can only be applied by Ph.Ds.

Some perfectly sound methods don’t work with practical programs.

- In theory, every variable has a unique name.
  - In practice, those names are hard to generate.
  - In practice, two variables can share a name.
  - In practice, those names can be so long that the expressions are unreadable.
- In theory, every program can be documented individually
  - In practice, there are lots of almost alike programs
  - In practice, “cut and paste” leads to errors
  - In practice, “cut and paste” documentation is boring to read and write.
- In theory, every program implements a mathematical relation
  - In practice, that function is hard to describe.

Software Quality Researchers Must Work with Industry

They can show us why theoretically correct methods won’t work.

They can provide us with insight-provoking examples.

They can contribute ideas, heuristics that can be made sound.

The cooperation must be two ways; we must give something in return.

- We can review their software.
- We can document aspects of their software.
- We can help them to review and document.
- We can inspect software.
- We can test software.

We can either tell them that we found no fault, or

We can find fault.

Either way, they gain.

Until we find no room for improvement in our approach, we will gain.

TOOLS and Use

The fact that the meaning of these tables has been precisely defined using classical mathematics means that we can make tools that support their use.

The methods are now used in safety-critical situations even though the tools are primitive.

With good tools and good education, we could turn software developers into Engineers.

Tools provide wonderful opportunities for Masters Degree research.

Cooperation with industry is needed to evaluate these tools.

Cooperation with industry is needed to “productise” these tools.

What’s New at UL for SQRL

SQRL is the obvious extension of my previous lines of research.

I have spent my life “swinging” between academia and industry.

Now, I can integrate both sides.

SFI has given me:

- professional staff for tool development
- professional staff for method exploration.

UL is providing me with

- colleagues with complementary interests and active research
- a very supportive and understanding environment.

Ireland and the Shannon area are providing me with cooperative industry.

- Tools will get used, criticised, and improved.
- Methods will get used, criticised, and improved.
- Students will get sound preparation and practical experience.

Dream:
The Irish Software Seal of Quality

The future in software will not go to those who build the cheapest software.

It will go to those who can:

- build the best software,
- provide trustworthy quality assessments of software built elsewhere,
- provide accurate and useful documents for software components.

This will require cooperation between researchers, educators and industry.

SQRL hopes that it can foster that cooperation.

Second Dream:
Irish Software Developer’s License Valued World-wide

Eventual SQRL Roles

These are very ambitious goals:

- a **research institute**, advancing the state of the art by developing new methods and tools,
- a **meeting ground** where SQRL researchers, other software researchers, and software developers can share their experience and ideas
- a **technology transfer institute**, helping industry to apply new research results and tools with video-conferenced seminars, short courses and advanced lectures as well as consultancy.
- a **software evaluation service**, preparing software quality assessments for software developers on a contractual basis and giving our supporters a competitive advantage.
- part of an **educational institution**, providing both industry and universities with highly qualified graduates who have a deep understanding of software development issues.