Software Engineering, Why And What

David Lorge Parnas[1]

Abstract

In the 1960s many of the scientists and mathematicians who were interested in computer software observed that software development was a profession that had more in common with Engineering than it did with the fields in which they had been trained. Most of them had mastered a body of knowledge and been taught how to extend that knowledge. They had not been taught how to apply what they had learned when building products that would be used by strangers.

It was proposed that, in addition to "Computer Science", there was a need for a for a new Engineering discipline called "Software Engineering". Although some critics considered the new field, redundant and superfluous, time has proven that it is needed. Today we depend on software in the same way that previous generations depended on traditional engineering products.

While the early pioneers correctly sensed that a new field was needed, they did not succeed in defining the capabilities required of practitioners in the new field. More recently, there have been several efforts to identify a "body of knowledge" for Software Engineering. In my opinion, none have captured the essence of the field.

This talk approaches the subject by focussing on capabilities, i.e., by addressing the question, What should a Software Engineer be able to do? Starting with two historical characterizations of the field by the pioneers, it presents a set of capabilities required by today's "Software Engineers".

---

[1] After lengthy discussions with Carl Landwehr, Jochen Ludewig Robert Meersman, Peretz Shoval, Yair Wand ,David Weiss and Elaine Weyuker whose contributions were substantial and substantive.

Contents

Why Talk About "Software Engineering"? (1)

In the beginning, there was Physics, Math, and Electrical Engineering

- Physicists were major computer users
- Mathematicians were interested in computer arithmetic, numerical methods, etc.
- Electrical Engineers built components and computers.
- There were also commercial users (with different computers!)
- Programs were frequently written for one's own use on one machine.
- Many people programmed but nobody built software.

Then came Computer Science with opponents and supporters!

- Supporters: Computer Science is a body of knowledge about computers and their use.
- Engineers: They won't learn Engineering (using science and math to design products).
- Mathematicians: They won't learn mathematics or how to use it.
- Scientists: There is no science; they won't learn how to organize the knowledge.
- Supporters: We can teach them all of that.
- With my "20 20" hindsight: The opponents were right!

Why Talk About "Software Engineering"? (2)

In the 60s, the title "Software Engineering" appeared.

- Scientists: We are not adding to the World's Knowledge
- Mathematicians: We are not proving or applying theorems!
- S&M in Chorus: It must be Engineering (whatever that may be).
- Engineers: You are not cutting tin (no physics). You have no discipline.

The famous NATO conferences (1968, 1969)

- Very few Engineers
- No agreement on what they were talking about!
  - Example: Not science because we worry about cost!
  - Example: They don't use math and neither do engineers.
- Created a field anyway - they got funding!
- Some opponents: It is just Computer Science
- Other opponents: It is just programming!

Two People Who Seem To Have Had Insight

Brian Randell:

- Programmer on late-early machines such as Ace (Turing)
- Early Algol compiler and book (real software)
- Many years at IBM research!
- Many years at Newcastle upon Tyne (GB)
- Headed many EU projects
- Told me something insightful - then forgot it!

Fred Brooks:

- Originally a physicist
- Worked with Howard Aiken at Harvard
- Published a “everything about computers book”.
- Went to IBM (development not research)
- Introduce and implemented the fundamental idea of a machine family with identical programmer’s manuals.
- The first really big Operating System - wrote about what he should not have done.

Randell's Insight

"Multi-person development of multi-version programs"

This implies all that differentiates software engineering from advanced programming.

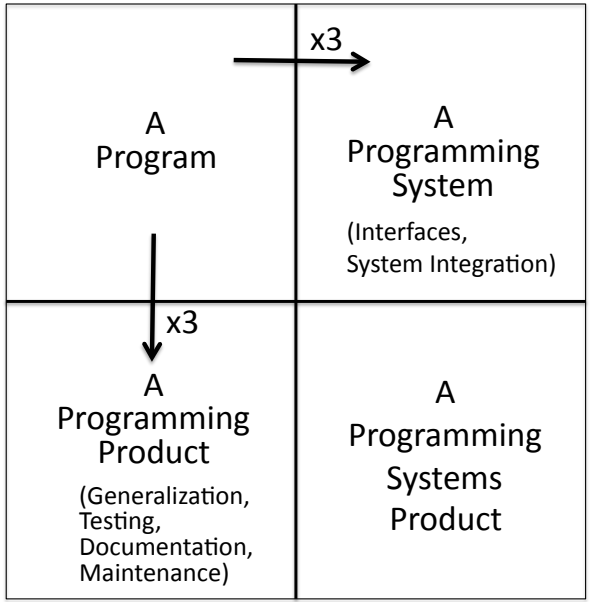
- **Software engineers must be able to work in teams.**
- **Software engineers produce programs that will be used, and often revised, by people other than the original developers.**
- **Software Engineers don't produce programs, they produce large families of programs.**

Performing that job professionally requires a mastery of programming, but many other capabilities are required as well.

This talk will discuss the required capabilities.

Fred Brooks's Insight

Beyond Programming: "productizing" and "integration":



Fred Brooks' explanation of why software engineering is more than programming.[2]

Much more effort than simple programming

- **Combining the work of many programmers**
- **Preparing the product for use by other people.**

[2] Figure redrawn from "The Mythical Man-Month". "x3", denotes a 3-fold increase in effort.

Randell And Brooks Essentially Agree, But….

It is much more than “learning to work with other people”.

It affects product design, analysis, and documentation.

It is deeper than specific tools, technologies, and platforms (technology).

Technology has been changing rapidly, and will continue to do so.

- Developers need principles that will last their careers.
- Designers must be taught how to use those principles.
- To learn to use principles, they must use those principles with guidance.
- To use principles, they must learn some technologies.
- Developers must learn the difference or they will be forever confused.

Use lectures, labs, and lab lectures.

- Lectures stress the principles.
- Lab lectures introduce the technology.
- Labs are where they practice under supervision.

What Do Software Engineers Need To Be Taught How To Do?

- Communicate precisely between developers and stakeholders
- Communicate precisely among developers
- Design human-computer interfaces
- Design and maintain multi-version software
- Design software for reuse
- Revise old programs
- Software quality assurance
- Develop secure software
- Create and use models in system development
- Specify, predict, analyze and evaluate performance
- Be disciplined in development and maintenance
- Use metrics in system development
- Manage complex projects
- Deal with concurrency
- Understand and use non-determinacy
- Apply mathematics to increase quality and efficiency.

Properties Of The Capabilities On The List

They are all “How to do it right” topics.

- “How to do it right” requires deep understanding.

They all involve some computer science.

They involve some mathematics

They involve understanding of science (knowledge organization)

They are not Science or Mathematics - they are something else.

They are fundamental - not technology.

Technological change will not invalidate this list.

Communication Between Developers And Stakeholders

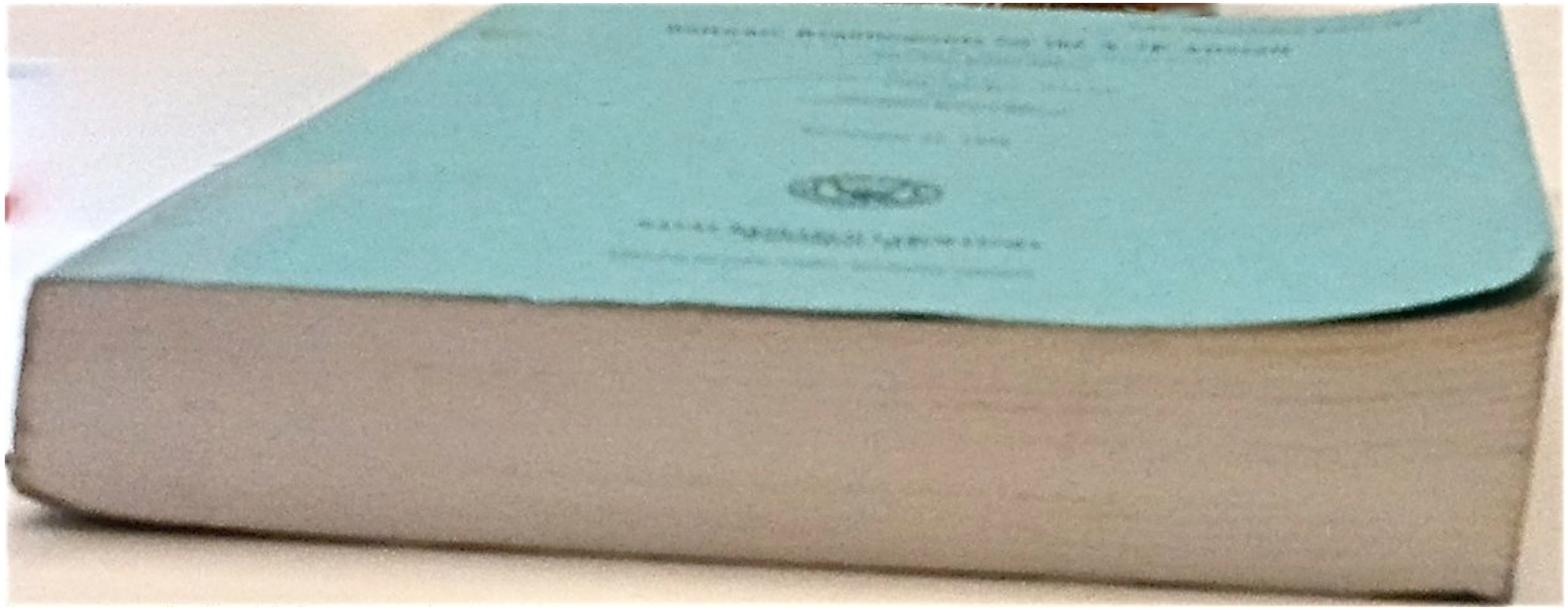
This communication can leads to lawsuits, loss of confidence.

- Completeness is important!
- Precision is important!
- Documentation is important!

There should be no user-visible decisions made by coders.

Know before you code!

How Much Information Is in a Requirements Document?

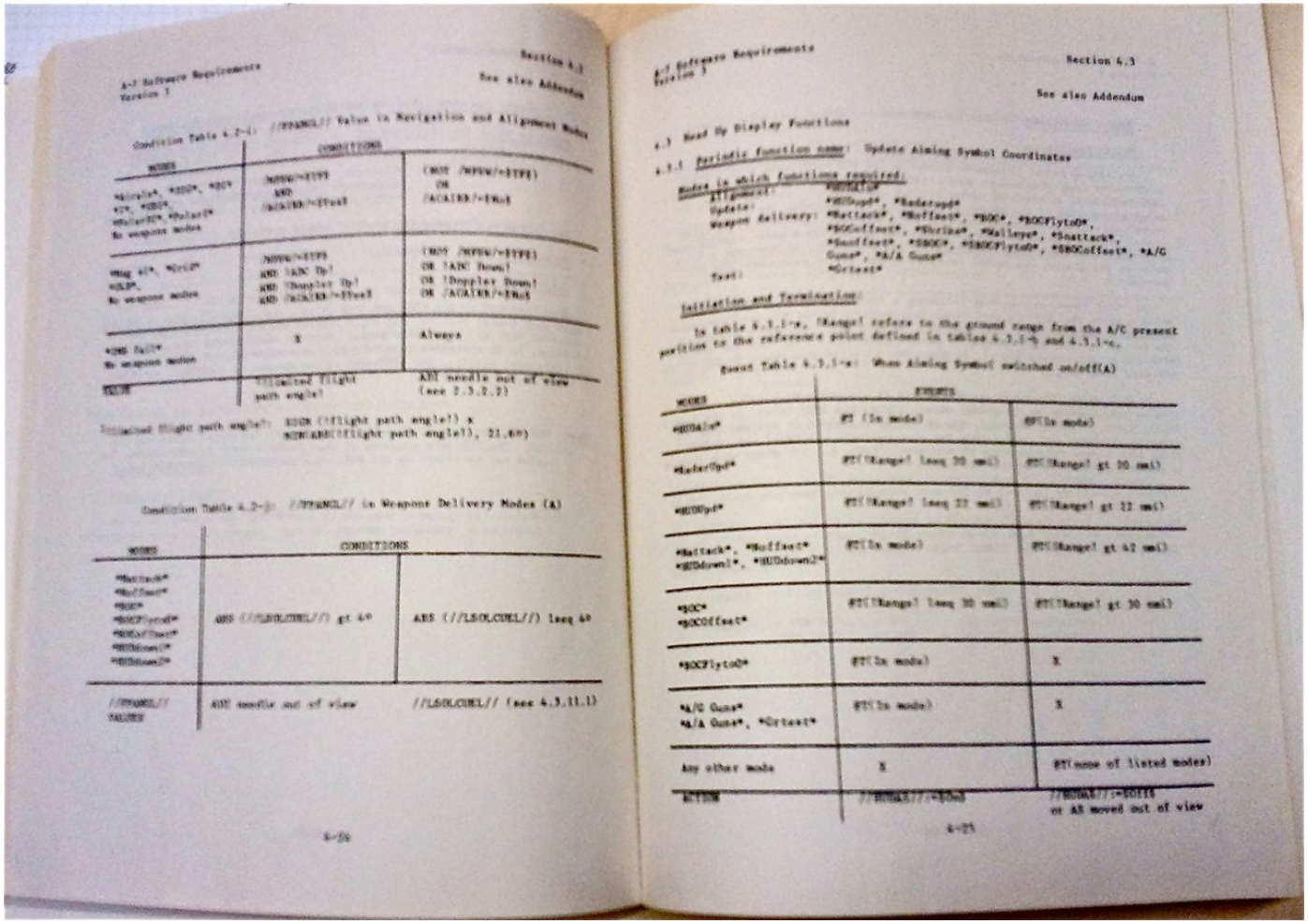


There is no code or data structure in this document!

Everything in this document can be observed by the user

Want to see inside?

A Typical Page



Why So Much Information?

We were writing requirements for programmers!

- We wanted to keep programmers from making user visible decisions.

The story of the “3500 feet” pull up queue.

- Moral: Programmers shouldn’t decide such things while coding!

A true story that is close to my heart (Pacemaker)

- Moral: Programmers shouldn’t decide such things while coding!
- Moral: Review by Subject Matter Experts is essential

Pilots and Engineers (not Software) found many errors in that document.

- The misinformation came from them. One person “knew it”, others corrected it.
- They could read it.

Surprise! The documentation for a component is often larger than the documentation for the complete system!

Communicate Precisely Among Developers

Working together to build software does not mean smiling and being helpful, positive, cooperative, etc., (not bad things of course).

- It requires effective communication of detailed information

Software construction by teams is organized as modules (assignments).

Each team member must know what is expected of his/her code.

Each programmer must know what can be expected of other modules.

Poor communication can leads to bugs that are hard to find/fix.

Some developers will be re-developers (maintainers) many years later.

- Completeness is important!
- Precision is important!
- Documents are important! (especially for maintainers).

A Keyboard Checking Component

Used on Production Lines

Well Documented

- Two Documents totalling 21 pages
- Several Errors
- Omissions
- Ambiguities

Could be replaced by two correct pages:

Keyboard Checking Main Pages

N(T) =

keyOK			T=_	¬ (T= _) ∧ N(p(T))=1	¬ (T= _) ∧ 1<N(p(T))< L	¬ (T= _) ∧ N(p(T))= L
				2	N(p(T))+ 1	Pass
¬keyOK ∧	¬keyesc ∧	(¬prevkeyOK ∧ prevkeyesc ∧ preprevkeyOK) ∨ prevkeyOK			N(p(T)) - 1	N(p(T)) - 1
		¬prevkeyOK ∧ prevkeyesc ∧ ¬preprevkeyOK			N(p(T))	N(p(T))
		¬prevkeyOK ∧ ¬ prevkeyesc	1	1	N(p(T))	N(p(T))
	keyesc ∧	¬ prevkeyesc		1	N(p(T))	N(p(T))
		prevkeyesc ∧ ¬prevexpkeyesc		Fail	Fail	Fail
		prevkeyesc ∧ prevexpkeyesc		1	N(p(T))	N(p(T))

Keyboard Checker - Other Page

Name	Meaning	Definition
keyOK	most recent key is the expected one	r(T) = N(p(T))
keyesc	most recent key is the escape key	r(T) = esc
prevkeyOK	key before the most recent key was expected one	r(p(T)) = N(p(p(T)))
prevkeyesc	key before the most recent key was escape key	r(p(T)) = esc
preprevkeyOK	key 2 keys before most recent key was expected key	r(p(p(T))) = N(p(p(p(T))))
prevexpkeyesc	key expected before most recent key was escape key	N(p(p(T)))= esc

This may not mean much to you but it should and could.

Remember that this replaces 21 pages full of problems.

This is almost all that you need to know to write or use the component.

This is checkable and can be used to improve testing.

Design Human-Computer Interfaces

Software Systems communicate with people.

To design a good interface, one must consider both:

- the nature of the information (including dynamic behaviour)
- the capabilities and habits of the users

Designers must balance ease-of-learning and ease-of-use.

A Multidisciplinary capability:

- psychology
- engineering biomechanics
  industrial design
  graphic design
- software design
- interface-device design

Improving user productivity and/or improving sales

Design And Maintain Multi-Version Software

Specific Issues:

- Separating (changeable) concerns
- Documenting to ease revision
- Using parameterization
- Design software that is easy to “port” to other platforms.
- Design software to be easily extended or contracted
- Managing products that exist in many versions

Separating (Changeable) Concerns

You have to learn to "*think change*". For most, it is not intuitive.

- **Study requirements to identify the aspects that are most likely to change.**
- **Study support system and hardware to identify the aspects that are most likely to change.**
- **Study software design decisions to identify those most likely to require change.**

Each module must be completely responsible for a changeable aspect.

Module interfaces that abstract from the changeable aspects.

- **Interfaces that are not likely to change when implementation is changed.**

45 years later, information hiding is still not well understood or applied.

- **Modules still do things but don't hide things.**

The answer is to habitually question, "Why won't it change?"

- **As simple as possible but not simpler**
- **Separation makes things simpler, but some things can't be separated.**

DocumentIng To Ease Revision

Developers forget why they document. It is not for them!

- Produce a “knowledge base” for future developers.
- Written rules for storage and retrieval.
- Organize the documentation so that information is easy to find.
- Information that is most likely to change should not be repeated.
- Maintain all artifacts (both code and documents) in an accurate and up-to-date state so that the documents can serve as a reliable source of information for future developers.

Poor standards are (usually) better than no standards.

Documentation standards need “teeth”.

- Johan’s prize-winning document.

Documentation standards need enforcers who think.

- An “elf” from the basement came up to about software safety.
  - The Halt and Set Fire Command is not used.
  - There are no sharp or cutting comments.

UsIng Parameterization

Remember “parameterized families of curves”?

- The formulas are the “commonalities”.
- The parameters are the variabilities.

Parameterize both documents and code.

- Parameters are the variabilities that you cannot hide.
- Interface parameters are in the specification.
- Same parameters are in the code.

Avoid including constants that are likely to change in their code.

Parameters do not have to be numbers.

- May be an enumerated type
- May be strings

Design Software That Is Easy To "Port" To Other Platforms.

Portability was not an issue when I entered this field.

- We tried to exploit our platforms as well as we could.
- We did not dream of moving our (machine dependent) code to a new machine.

Today, portability is increasingly important.

- People have several platforms.
- They expect to be able to move freely between them
- They are used to getting new ones every few years.

Portability is not easy. There are deep conflicts.

- Users want same "look and feel" as other applications on their platform.
- Users want product to have the same "look and feel" on all of their platforms.
- One platform may support services that not offered on some other platforms.
- Mechanisms available for transferring data between applications differ.
- Error handling conventions can differ between platforms.

Companies screw up their products to enhance platform independence.

Design Software To Be Easily Extended Or Contracted

Extension: Adding new services without changing existing code.

- Old data and functions continue to be useful.

Contraction: Removing services without changing remaining code.

- Contracted programme can be offered as free “bait” for its extensions,
- Data can be transferred to smaller platforms.

Software built this way is cheaper to maintain.

Compatibility can be guaranteed.

Products less frustrating than “almost alike” product lines.

Important to understand “uses hierarchy” vs. other structures.

Good development strategy

- Meet deadlines with useful, upward compatible, subset.
- Gain maturity for lower levels while developing upper levels.

Managing Products That Exist In Many Versions

Software organizations maintain many versions of their software products.

There are many components; several variations of each may exist.

It can be very difficult to make sure that a particular version of the product contains the right version of each of its components.

This is often called configuration management.

Software Engineers must learn how to manage a product's many files.

There are many configuration management tools available to organize the versions of components and assemble working systems.

The "architecture" of the software can make a huge difference in the difficulty of configuration management.

Software developers must learn to design with this problem in mind.

Design Software For Reuse

Software reuse is like good parenting.

- Everyone is in favour of it, but it is often very difficult.

Software must be designed, and documented, with future reuse in mind.

- Every component must be something worth reusing.
- It is often harder to reuse a component than to produce a new one.

When an organization has reusable code,

- The developers of a new product may not know about it.

It's not just code that we should reuse.

- Interface designs (the more reusable the interface, the better it is).
- Documentation (Don't attempt document reuse by conglomeration).
- Test Suites

This too is more easily said than done.

Revising Old Programs[3]

We have all been taught to write programs “from scratch”.

That is almost never the job!

We need to

- understand old code (even our own),
- leave code with better structure than it had when we found it,
- maintain or improve conceptual integrity
- make the revised code easier to maintain than the previous version.

[3] I am grateful to Dr. Prokop of Allianz Versicherung for his thoughts on this issue.

Software Quality Assurance

Software written for others must be better than programs for our own use.

- I can recover from failures of my own code but not your code.
- Ignorant users need robust software.

There are several complementary approaches to Software QA.

- testing programs — both to find and eliminate faults and to estimate reliability
- inspection of large programming systems (divide and conquer)
- formal verification of critical small programs, and
- reviewing documentation for formal completeness and accuracy.

There is a huge amount of knowledge that can be practical for developers.

Proper design makes SQA easier and more effective.

Develop Secure Software

Computer Security is our biggest challenge.

It is not an “add on” feature.

It is far more than encryption (the most mathematically tractable problem).

It is not just network design; networks make it more important and harder.

It is not just OS design; “Trojan horses” can be disguised as anything.

Security must be considered with every design decision.

Dijkstra’s “Weakest Precondition” must be taught as a security concept.

We have to apply “wp” and “wlp” throughout our design.

It is more a way of thinking than a formalism but both help.

Denial of service must always be on our minds.

Use Of Models In System Development

"Model" is one of many CS buzzwords and fads.

- Can you define it?
- How do you tell a good model from a bad one?

Models (physical, mathematical, or diagrammatic) essential in Engineering

- They are simplified versions of a product or component.
- Simplification has its cost. Models are often lies.
- Ohm's law (E = IR) is a model. It is not true. Errors follow from believing it.

Great care must go into creating a model.

Even greater care is required when interpreting an analysis of a model.

Modelling is not programming; it is far more general.

It is not CS; it is far more general.

Undefined Muddling Languages are harmful. Method is not language

Performance Specification, Prediction, Analysis and Evaluation

Performance of complex systems hard to predict

- Users and other external factors hard to predict
- Resource contention hard to predict.

Classical methods developed outside of CS

Classical methods have proven useful for computer systems.

Specifications/requirements must characterize usage patterns.

Specifications/requirements must be conditional.

Discipline In Development And Maintenance

Engineering specialities are called “disciplines” for many reasons

- Engineers are often required to follow rigid processes. They are enforced.
- Process may be embodied in forms to completed (measurements, calculated values, etc).
- Repairs and revisions are also subject to these disciplines.

Self-styled “Software Engineers” have no such discipline.

Engineers are not born with disciplined work habits; they have to be taught.

- Maintenance that maintains conceptual integrity (must be documented)
- Software maintenance requires at least as much discipline as the original development.
- A standard development process, including work products that must be produced, helps.
- Fake it[4] when it fails.
- Standards must be “falsify-able”
- Standards must be enforced, thoughtfully and using inspections.

---

[4] 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, pp. 251-257

Use Of Metrics In System Development

Engineers have long used “figures of merit” to compare products.

In software, they are usually called “metrics”.

Many software metrics mislead - they do not measure what they should.

- Better numbers may not mean better product.
- A developer can often “game” the metric - improving it without improving the quality.

Productivity metrics are particularly questionable.

- Programmers who solve a problem in 50 lines seem less productive than those who use 100.

Metrics must be used with caution but they will be used.

Their use needs to be taught.

Dealing With Concurrency

Concurrency is a theoretical problem in CS but a real problem in SE

There are three distinct concurrency design problems.

- Decomposition - How do you divide work into process?
- Synchronization: How do you prevent processes from cooperating illogically?
- Scheduling: How to make things happen on time and in accord with priority requirements.

Decomposition is the most important task but theoreticians overlook it.

- Synchronization is easier with the right decomposition.
- Scheduling gets easier with the right decomposition.

Do not try to solve priority problems with synchronization.

Software Engineers should know how to do these things. They do not!

They often get the decomposition wrong and cannot solve the others as a result.

Using Non-Determinacy

Non-determinacy is a theoretical problem in CS but a practical tool for SE.

Some things are visible but not important.

- **We should not over-constrain solutions or make unwarranted assumptions.**

Some things are unpredictable but can be dealt with.

- **To deal with the unpredictable we must know what the possibilities are.**
- **Example: Are strings left or right justified?**

Satisfying constraints that do not fully determine the answer.

- **US Corporate Tax Law for multi-national corporations**
- **Bob Floyd's non-determinacy (choice of order of search).**

The Software Engineer who understands the true meaning of non-determinacy can design much better software (especially networks).

Managing Complex Projects

Coordinating the work of many people is a complex task. It requires:

- **planning and scheduling,**
- **estimation of cost, time and effort needed for a task,**
- **progress measurement,**
- **problem tracking,**
- **risk management,**
- **configuration management,**
- **resource control,**
- **task assignment,**
- **forming teams and other organizations (e.g. matrix organizations),**
- **choosing and using project management tools**
- **leadership**

It is not Computer Science, but all are relevant in multi-person projects.

Two Views of Project Management and Software Engineering:

Project management is an important problem for all engineering disciplines.

- Project management is often taught by both Management and Engineering faculties.
- Much of what is taught is independent of the nature of the product.
- Consequently, project management is outside of Software Engineering.
- Can be a one-year "add on" for all Engineering disciplines.

Project management is more difficult in software projects.

- Problems arise in software projects that are not common in other disciplines.
- Consequently, project management is integral to Software Engineering.

Bad design, poor documentation, weak discipline make management hard.

Software project management is difficult with incomplete information.

- Incomplete information is most noticeable when requirements are poorly documented.
- Interface documentation can make management easier. (Mythical Man Month effect)

When we master design and documentation, management will be more normal but it will always be an important capability.

Mathematics In Software Engineering

Mathematics is an essential tool for all Engineers.

- Not a topic for its own interest (although very interesting)
- Know how to use it more than how to grow it.
- Applied Math is the model, not logic or other “pure math”.

Do not confuse representation of abstractions with the abstraction.

Use the basic ideas, not the newest or most advanced.

Beware the “Every educated man … ” arguments.

Use it from the start when teaching - it is not an add-on.

- Specify assignments precisely
- Use math when making explanations/correctness arguments.

Don’t let logicians teach the logic.

- They want to prove theorems.
- Engineers need to describe and analyze products.

The Role of Lab Work in Software Engineering Education

To learn to do, you must do.

You must not confuse technology with design principles.

Consequently, both must be taught but they should be taught separately.

Competitive projects (every team building the same product) works well.

- Round Robin recruitment to form equally competitive teams.
- Pre-competitive phase to set basic requirements and criteria.
- Multi-league organization increases value of documentation.
- Industrial judges increase motivation (often donate prize).

Projects must show the advantages of the approaches being taught.

- Initial SE course uses pair (writer/tester) organization to show advantage of having specs.
- Teaches reading docs before teaching writing docs.
- Docs save time, reduce disagreements. That should be experienced in labs.

View This Talk as a Capabilities Checklist

Use it and extend it.

- Such a list has long been needed.
- It is not a BOK. it is a CBOC (Core Body of Capabilities) for Software Engineers

This is not a jurisdictional dispute.

We are not competing for jobs or students.

We are trying to do two jobs well.

- preparing students for the career they want
- training people for the jobs that need to be filled

Are you building software or doing exploratory research/design?

Summary (1)

Software Engineering is not a subset of Computer Science.

- There is much of what is relevant is not taught in Computer Science programmes
- There is much to learn that is not as relevant for computer scientists.

Software Engineering is not a superset of Computer Science.

- There is much computer science that is irrelevant for software developers.

We need both!

We probably need more SEs than CSists.

Brooks and Randell were right and have been ignored for too long.

This talk is a checklist. - Use it and extend it.

Think hard about what you are, what you teach, and who you hire.

- Is the job engineering, passive science, or research?

Summary (2)

Science and Engineering are fundamentally different activities.

- Students often need help to make the right choice.

Engineering and Technology are fundamentally different.

- Students often need help to make the distinction.
- Some people are only interested in technology. We need them too.

Projects need to be designed to teach specific points.

- Otherwise it is often just coding experience.

Professional responsibility of Engineers is important

- Quality Assurance must be viewed as an obligation.
- Safety and Security must be viewed as an obligation.

Every (future or present) SE needs an overview (course or lectures).

Some Questions For You!

If you are a developer, do you have all these capabilities?

- If not, consider professional development courses.

If you are a manager, do your team members know how to do these things?

- Consider courses to upgrade your staff.

If you are a teacher, does your programme teach these things?

- Review your curriculum - perhaps establish some new ones

Software Engineers need these capabilities, Computer Scientists (if they really are scientists) can usually get along without them.

The differentiator is "Multi-Person construction of Multi-Version programs".

- Are you producing something that others will be using?