



# Excavating the knowledge of the ancestors

What we can learn today from the lost wisdom of earlier days

Uwe Friedrichsen (codecentric AG) – Berlin Expert Days – Berlin, 21. September 2017

# Uwe Friedrichsen

IT traveller.

Connecting the dots.

Attracted by uncharted territory.

CTO at codecentric.

<https://www.slideshare.net/ufried>

<https://medium.com/@ufried>



@ufried

IT these days ...

A photograph of a modern building with a curved, reflective metallic facade. The building's surface is highly polished and reflects the sky and surrounding environment. Several windows are visible, some of which are slightly open. The overall aesthetic is sleek and futuristic.

An IT solution from the outside ...

A dark, cluttered interior space, possibly a basement or a storage room, filled with various objects and debris. The scene is dimly lit, with a blueish tint. In the foreground, a wooden ladder leans against a wall. To the left, a globe sits on a stand. In the center, a mouse is visible near a yellow clock. To the right, a red and blue bicycle is partially visible. The floor is covered with various items, including a green bucket, a yellow can labeled 'PEARS', a broom, and several pairs of shoes. The walls are covered with various items, including a mounted animal skull, a hat, and a framed picture. The overall atmosphere is one of a neglected and chaotic space.

... and from the inside

And how do we address this situation?

Let us add some new & cool stuff ...





... as anything old is useless crap





Still, some treasuries can be excavated

Opening thought

**Computation and State Machines**

Leslie Lamport

19 April 2008

# Computation and State Machines

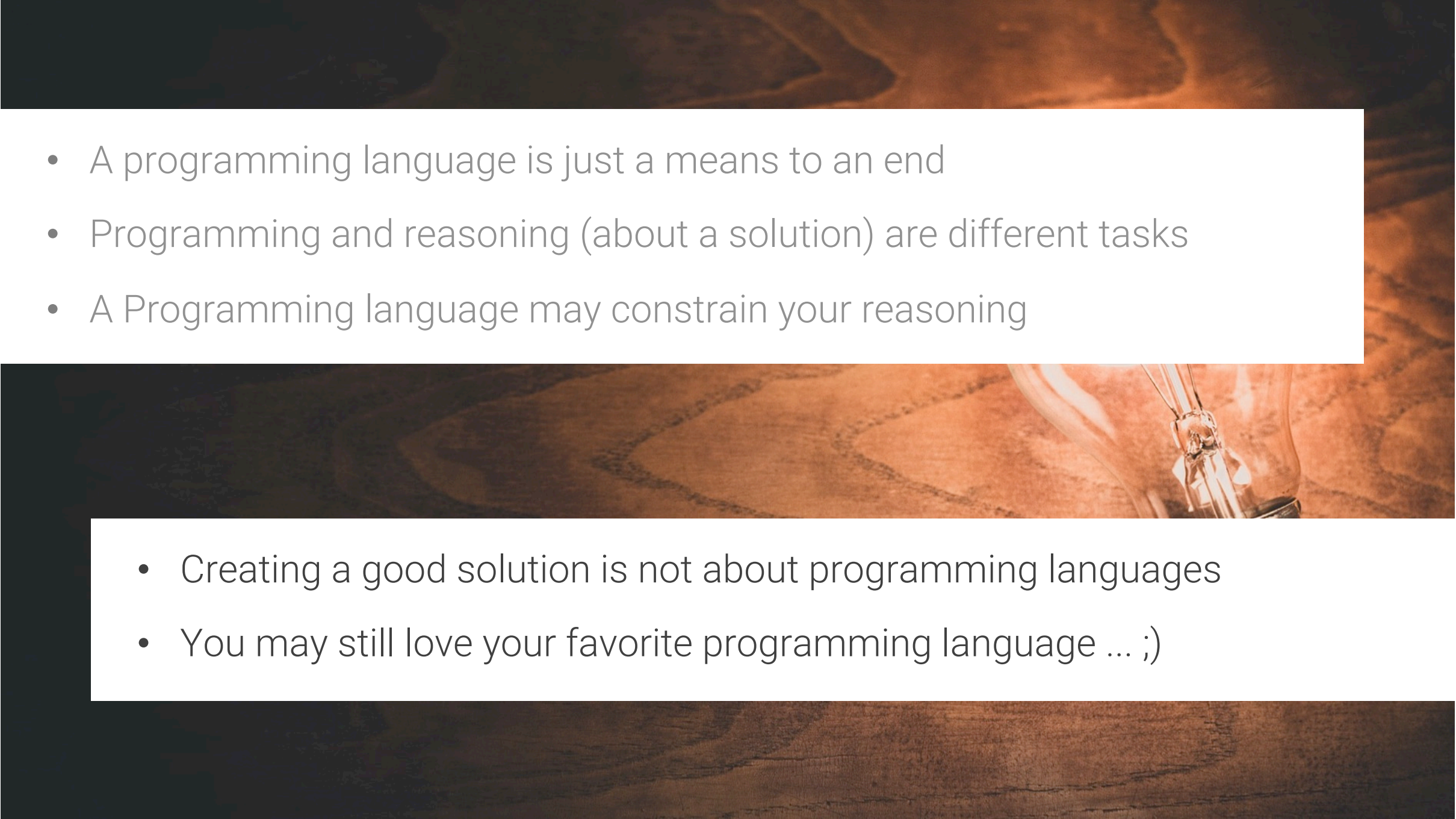
by Leslie Lamport

[Lam 2008]

“For quite a while, I’ve been disturbed by the emphasis on language in computer science. **One result of that emphasis is programmers who are C++ experts but can’t write programs that do what they’re supposed to.**

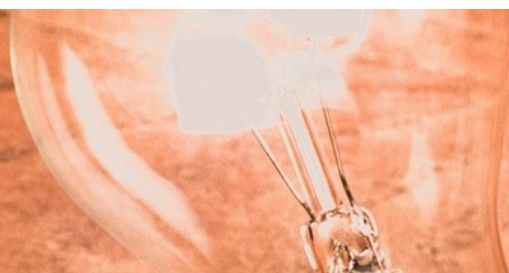
The typical computer science response is that programmers need to use the right programming/specification/development language instead of/in addition to C++. The typical industrial response is to provide the programmer with better debugging tools, on the theory that we can obtain good programs by putting a monkey at a keyboard and automatically finding the errors in its code.

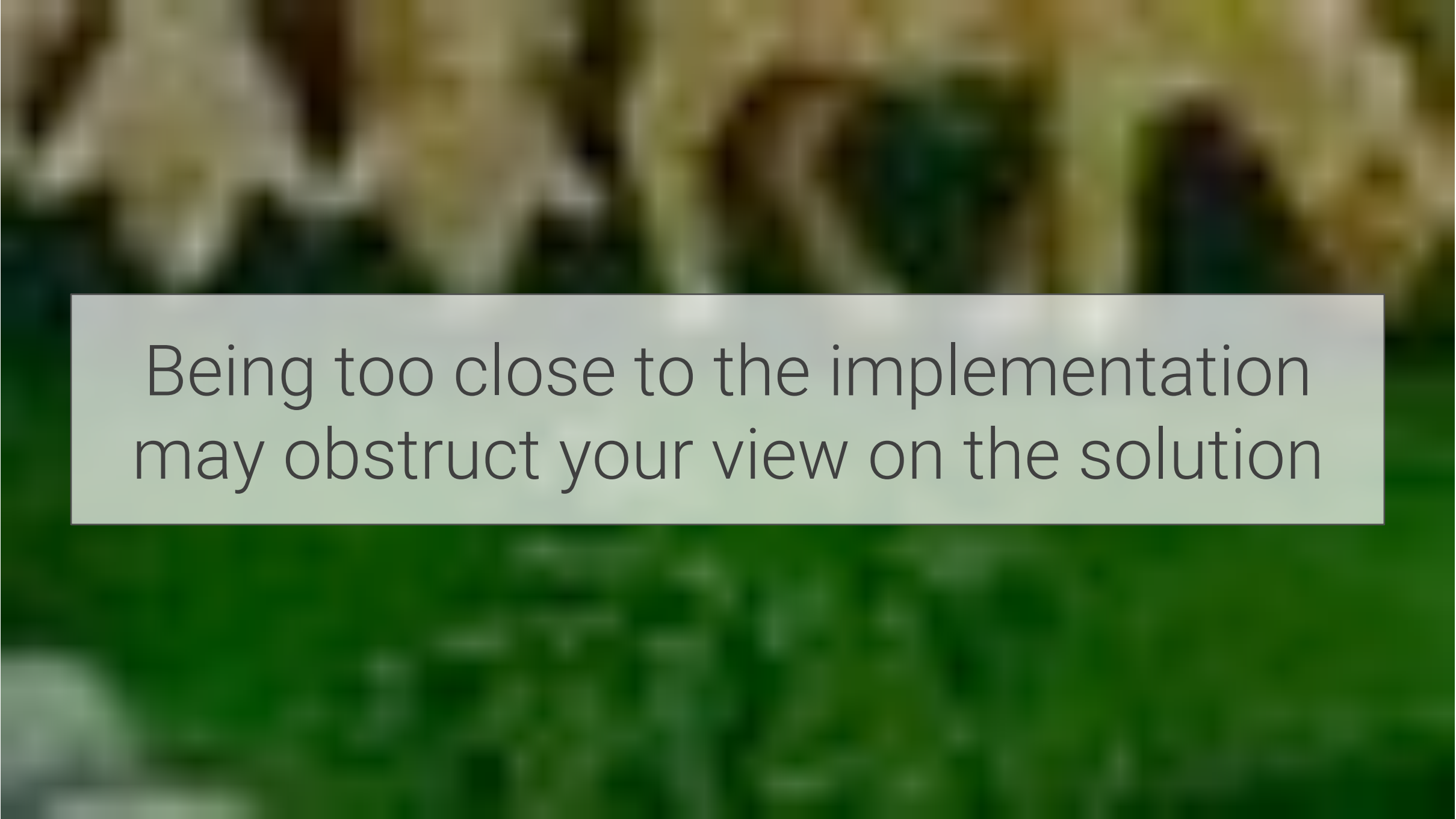
**I believe that the best way to get better programs is to teach programmers how to think better. Thinking is not the ability to manipulate language; it’s the ability to manipulate concepts.** Computer science should be about concepts, not languages.”

- 
- A lightbulb is shown in the lower right corner of the slide, resting on a wooden surface. The background is a warm, textured wood grain. The lightbulb is unlit, but its presence suggests a concept or a solution.
- A programming language is just a means to an end
  - Programming and reasoning (about a solution) are different tasks
  - A Programming language may constrain your reasoning

- Creating a good solution is not about programming languages
- You may still love your favorite programming language ... ;)

- The best programming language will not fix a poor design
- Debuggers and other tools will not create good solutions magically

- 
- Creating a good solution is about understanding the domain first, then a lot about reasoning (finding abstractions, interfaces, etc.)
  - Do not rely on tools to solve problems – rely on our skills



Being too close to the implementation  
may obstruct your view on the solution



You can't see the whole picture in your IDE



System and interface design

October 1995

## Go To Statement Considered Harmful

*Edsger W. Dijkstra*

---

Reprinted from *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148.  
Copyright © 1968, Association for Computing Machinery, Inc.

This is a digitized copy derived from an ACM copyrighted work. It is not guaranteed to be an accurate copy of the author's original work.

---

**Key Words and Phrases:**

go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

**CR Categories:**

4.22, 6.23, 5.24

**Editor:**

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For

# Go To Statement Considered Harmful

by Edsger W. Dijkstra

[Dij 1968]

“My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. [...]

My second remark is that **our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed**. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.”

What we actually try to do

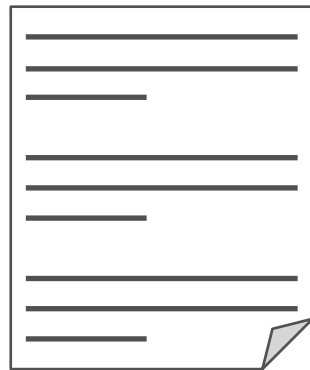
Reason



Developer

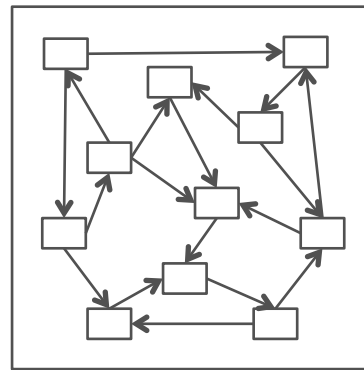
Create

Reason



Source Code

Create



Runtime Process

Create



Value

What we need to do instead

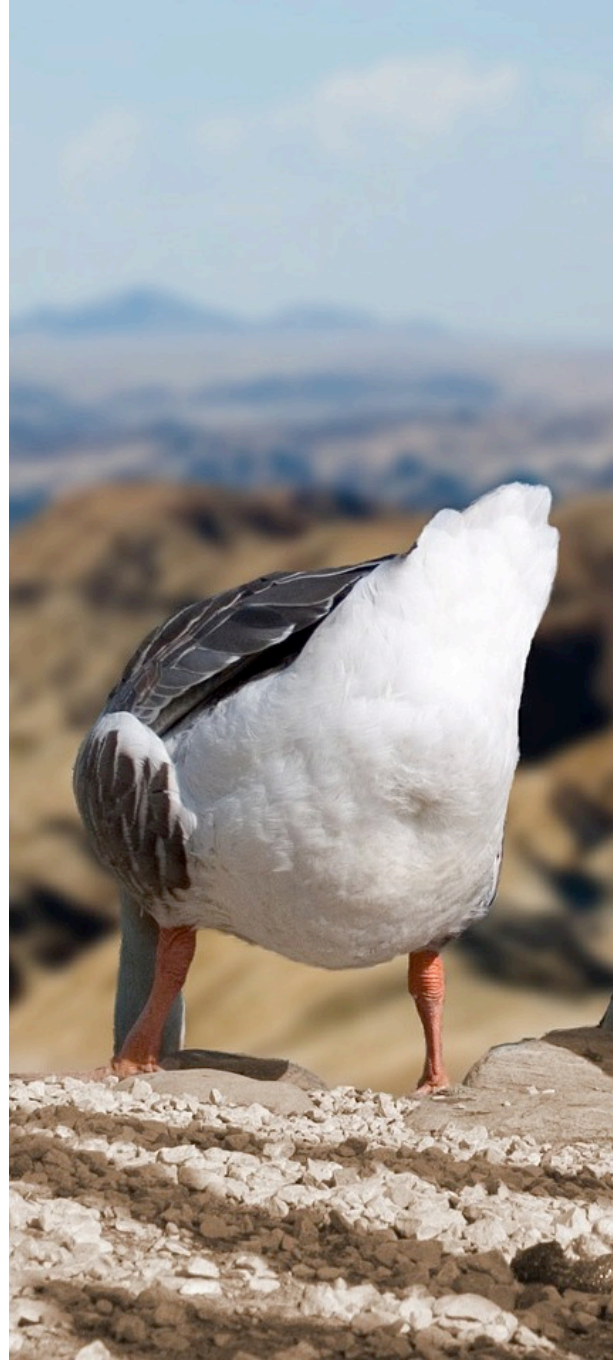
- This needs to be
- **as comprehensible as possible** and
  - **support us in reasoning about runtime behavior**

We are poor in understanding that

That creates the (customer) value

# Insights for today

- Make your design and code as understandable as possible
- Reason about a good and comprehensible modularization
- Reason about module groupings and hierarchies
- Create egoless code
- Augment with additional documentation where helpful
- Always have the reader of your design and code in mind



## On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas  
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

**Key Words and Phrases:** software, modules, modularity, software engineering, KWIC index, software design

**CR Categories:** 4.0

### Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gauthier and Pont [1, §10.23], which we quote below:<sup>1</sup>

A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

### A Brief Status Report

The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above.

<sup>1</sup>Reprinted by permission of Prentice-Hall, Englewood Cliffs, N.J.

Copyright © 1972, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

On the criteria to be  
used in decomposing  
systems into modules

by David L. Parnas

[Par 1972]

“The effectiveness of a ‘modularization’ is dependent upon the criteria used in dividing the system into modules.”

“The second decomposition was made using "information hiding" as a criterion. [...] **Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.**”

“There are a number of design decisions which are questionable and likely to change under many circumstances. [...] By looking at these changes we can see the differences between the two modularizations.”

## Separation of concerns

One concept/decision per module



## Information hiding

Reveal as little as possible about  
internal implementation



## Better changeability

Changes are kept local

## Independent teams

Teams can easier work  
independently on different modules

## Easier to comprehend

Modules can be understood on  
their own easier

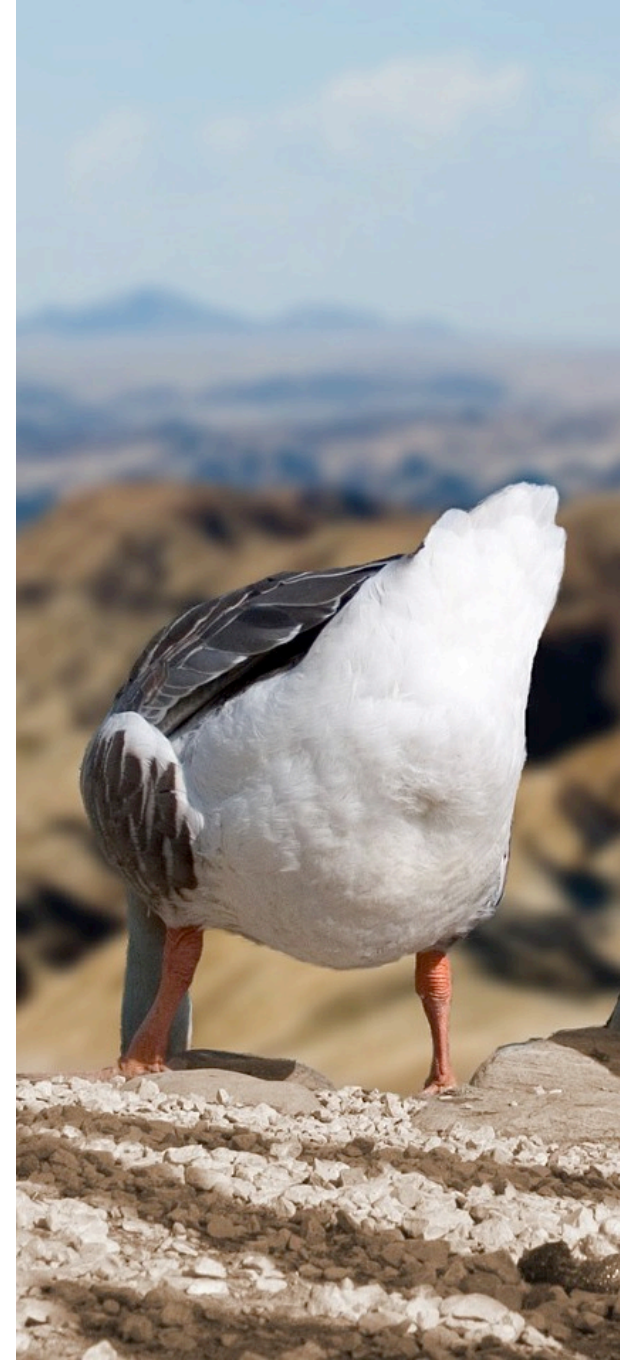


“If we give that title a slight twist –  
‘On the criteria to be used in decomposing systems into **services**’ –  
it’s easy to see how this 45-year old paper  
can speak to contemporary issues.”

<https://blog.acolyer.org/2016/09/05/on-the-criteria-to-be-used-in-decomposing-systems-into-modules/>

# Insights for today

- Enforce Separation of concerns
  - Understand the main change drivers to identify the appropriate concerns to be encapsulated
  - Still, that is not the only encapsulation criterion
- Work hard to provide a minimal interface
  - Client-driven contracts can help
  - “Less is more”



1971

## Information distribution aspects of design methodology

David Lorge Parnas  
*Carnegie Mellon University*

Follow this and additional works at: <http://repository.cmu.edu/compsci>

This Technical Report is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Computer Science Department by an authorized administrator of Research Showcase @ CMU. For more information, please contact [research-showcase@andrew.cmu.edu](mailto:research-showcase@andrew.cmu.edu).

# Information distribution aspects of design methodology

by David L. Parnas

[Par 1971]

**“The connections between modules are the assumptions which the modules make about each other.** In most systems we find that these connections are much more extensive than the calling sequences and control block formats usually shown in system structure descriptions.

We now consider making a change in the completed system. [...] We may make only those changes which do not violate the assumptions made by other modules about the module being changed. In other words, **a single module may be changed only as long as the ‘connections’ still ‘fit’.** Here, too, we have a strong argument for making the connections contain as little information as possible.”

# The 1980 ACM Turing Award Lecture

Delivered at ACM '80, Nashville, Tennessee, October 27, 1980



C.A.R. Hoare

The 1980 ACM Turing Award was presented to Charles Antony Richard Hoare, Professor of Computation at the University of Oxford, England, by Walter Carlson, Chairman of the Awards committee, at the ACM Annual Conference in Nashville, Tennessee, October 27, 1980.

Professor Hoare was selected by the General Technical Achievement Award Committee for his fundamental contributions to the definition and design of programming languages. His work is characterized by an unusual combination of insight, originality, elegance, and impact. He is best known for his work on axiomatic definitions of programming languages through the use of techniques popularly referred to as axiomatic semantics. He developed ingenious algorithms such as Quicksort and was responsible for inventing and promulgating advanced data structuring techniques in scientific programming languages. He has also made important contributions to operating systems through the study of monitors. His most recent work is on communicating sequential processes.

Prior to his appointment to the University of Oxford in 1977, Professor Hoare was Professor of Computer Science at The Queen's University in Belfast, Ireland from 1968 to 1977 and was a Visiting Professor at Stanford University in 1973. From 1960

to 1968 he held a number of positions with Elliot Brothers, Ltd., England.

Professor Hoare has published extensively and is on the editorial boards of a number of the world's foremost computer science journals. In 1973 he received the ACM Programming Systems and Languages Paper Award. Professor Hoare became a Distinguished Fellow of the British Computer Society in 1978 and was awarded the degree of Doctor of Science *Honoris Causa* by the University of Southern California in 1979.

The Turing Award is the Association for Computing Machinery's highest award for technical contributions to the computing community. It is presented each year in commemoration of Dr. A. M. Turing, an English mathematician who made many important contributions to the computing sciences.

---

## The Emperor's Old Clothes

Charles Antony Richard Hoare  
Oxford University, England

---

**The author recounts his experiences in the implementation, design, and standardization of computer programming languages, and issues a warning for the future.**

**Key Words and Phrases:** programming languages, history of programming languages, lessons for the future  
**CR Categories:** 1.2, 2.11, 4.2

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.  
Author's present address: C. A. R. Hoare, 45 Banbury Road, Oxford OX2 6PE, England.  
© 1981 ACM 0001-0782/81/0200-0075 \$00.75.

My first and most pleasant duty in this lecture is to express my profound gratitude to the Association for Computing Machinery for the great honor which they have bestowed on me and for this opportunity to address you on a topic of my choice. What a difficult choice it is! My scientific achievements, so amply recognized by this award, have already been amply described in the scientific literature. Instead of repeating the abstruse technicalities of my trade, I would like to talk informally about myself, my personal experiences, my hopes and fears, my modest successes, and my rather less modest failures. I have learned more from my failures than can ever be revealed in the cold print of a scientific article and now I would like you to learn from them, too. Besides, failures

Communications  
of  
the ACM

February 1981  
Volume 24  
Number 2

# The emperor's old clothes

by Sir Charles Antony Richard Hoare

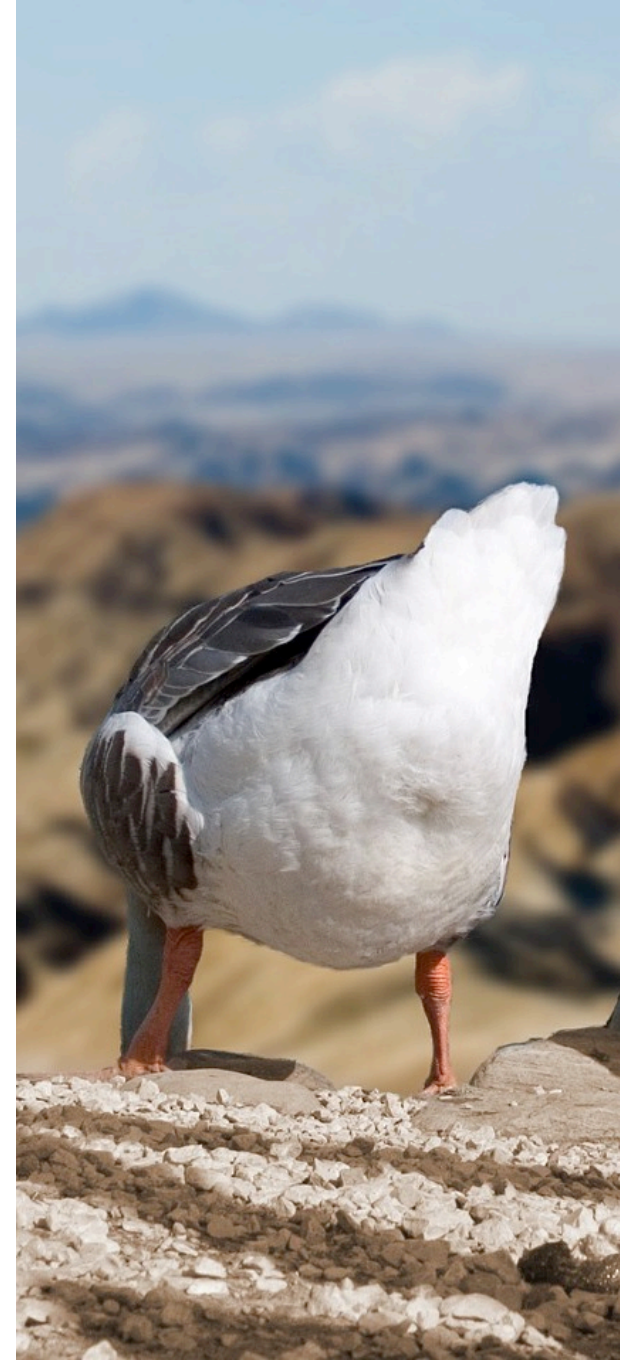
[Hoare 1981]

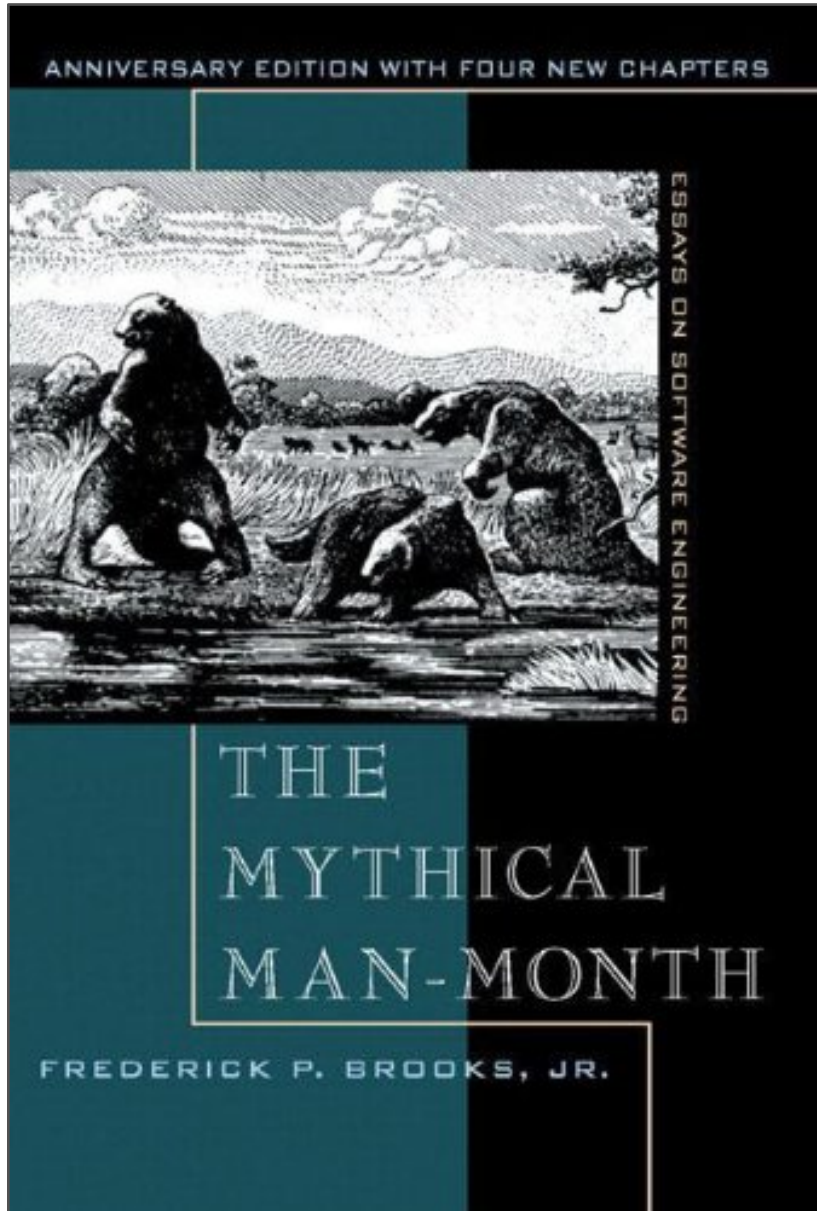
“I conclude that there are two ways of constructing a software design: **One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.** The first method is far more difficult.

At first I hoped that such a technically unsound project would collapse but I soon realized it was doomed to success. Almost anything in software can be implemented, sold, and even used given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars. **But there is one quality that cannot be purchased in this way - and that is reliability. The price of reliability is the pursuit of the utmost simplicity.** It is a price which the very rich find most hard to pay.”

# Insights for today

- It is easy to create an overly complex solution, but it is very hard to create a simple solution
- Reliability requires simplicity
  - Work hard for easy to grasp concepts
  - Do not confuse “simple” with YAGNI
- “Everything should be made as simple as possible, but not any simpler” – Albert Einstein
- Make it hard to misuse your solution / design / API





## The tar pit

by Frederick P. Brooks, Jr.

(taken from the "The mythical man-month")

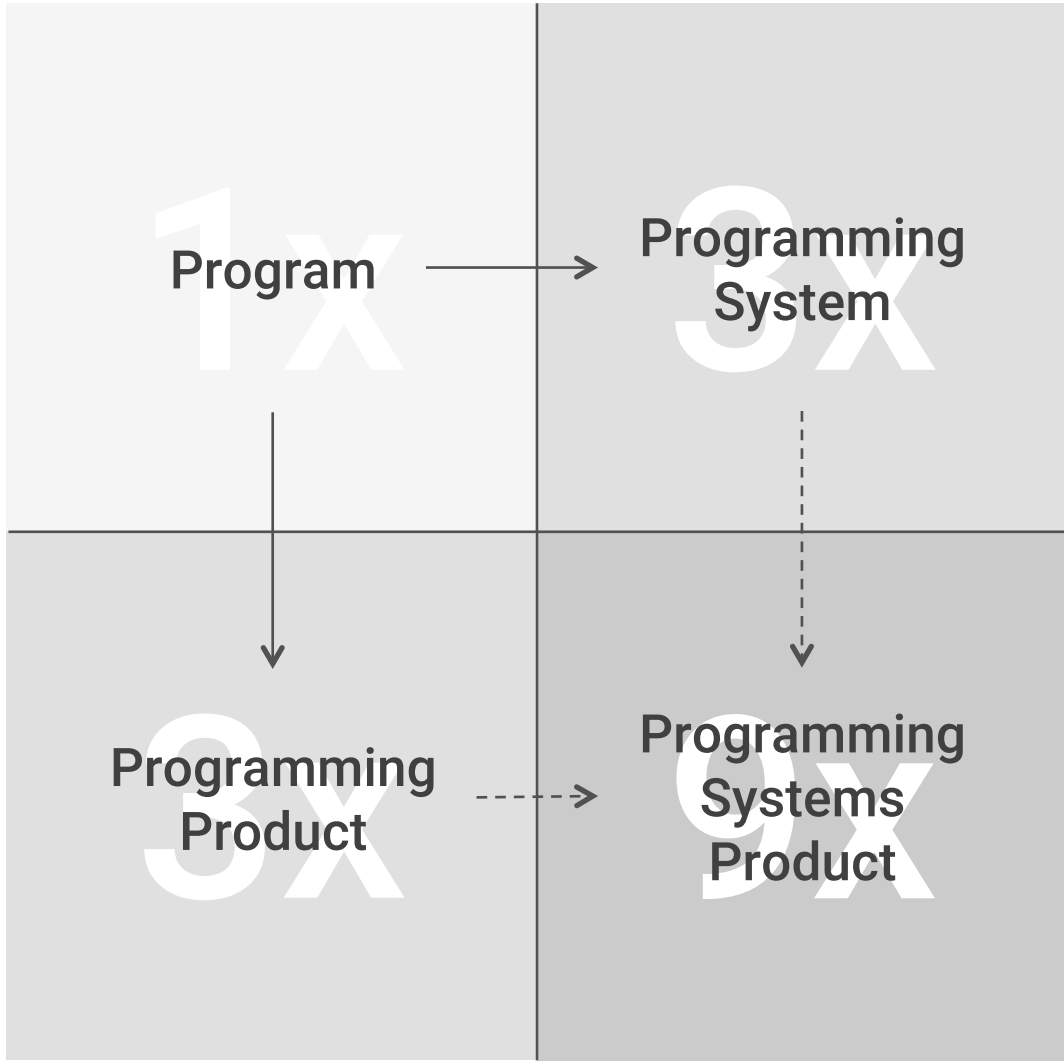
[Bro 1995]



“One occasionally reads newspaper accounts of how two programmers in a remodeled garage have built an important program that surpasses the best efforts of large teams. And every programmer is prepared to believe such tales, for he knows that he could build any program much faster than the 1000 statements/year reported for industrial teams.

Why then have not all industrial programming teams been replaced by dedicated garage duos? One must look at what is being produced.”

The original module,  
suitable for the context  
it was created for



A module, ready to be  
used in an ecosystem  
of interacting modules

A generalized module,  
suitable for multiple  
contexts

A (re-)usable module,  
that provides a general  
solution for a problem

**1x**  
**Program**

Completeness of accessibility

- Precise interface definition
- Clear behavioral contract
- Thorough integration (or alike) testing
- API Documentation

Completeness of functionality

- Hardening implementation
- Handling of edge cases
- Thorough testing
- Design Documentation

**9x**  
**Programming**  
**Systems**  
**Product**

Replace the following terms in your mind:

- **Program** with **(Micro)Service**
- **Programming Product** with **Robust Service**
- **Programming System** with **Robust API**
- **Programming Systems Product** with **Robust and re-usable Service incl. API**

Now you should see when it is worth going the whole 9 yards

# Wrapping up

## System and interface design

- Make your design and code as understandable as possible
- Enforce Separation of concerns
- Work hard to provide a minimal interface
- Reliability requires simplicity
- Creating a robust service requires a lot of hard work
- Creating a good API requires a lot of hard work



# Options and trade-offs

 **TANDEM** COMPUTERS

**The 5 Minute Rule for Trading  
Memory for Disc Accesses and  
the 5 Byte Rule for Trading  
Memory for CPU Time**

Jim Gray  
Franco Putzolu

Technical Report 86.1  
May 1985, Revised February 1986  
PN87615

## The 5 minute rule

by Jim Gray and Franco Putzolu

[Gra 1986]

“One interesting question is: When does it make economic sense to make a piece of data resident in main memory and when does it make sense to have it resident in secondary memory (disc) where it must be moved to main memory prior to reading or writing?”

**Pages referenced every five minutes should be memory resident.**

The 80-20 rule implies that about 80% of the accesses go to 20% of the data, and 80% of the 80% goes to 20% of that 20%. So 64% of the accesses go to just 4% of the database. **Keeping that 4% of the database in the main memory disc cache saves 64% of the disc accesses over the all-on-disc design. [...]** This is a net 270K\$ savings over the all-on-disc design and a 1.27M\$ savings over the all-in-main-memory design.”



**The Five-Minute Rule Ten Years Later,  
and Other Computer Storage Rules of Thumb**

Jim Gray  
Goetz Graefe  
September 1997

Technical Report  
MSR-TR-97-33

Microsoft Research  
Microsoft Corporation  
One Microsoft Way  
Redmond, WA 98052

This paper appeared in the SIGMOD Record 28(4): 43-48 (1997).  
The original document is at [http://research.microsoft.com/~grayj/tech\\_rpts/msr\\_tr97\\_33.pdf](http://research.microsoft.com/~grayj/tech_rpts/msr_tr97_33.pdf).

ACM  
Copyright ©1996 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM Inc., 351 (312) 898-9481, or [permissions@acm.org](mailto:permissions@acm.org).

# The 5 minute rule 10 years later

- 5 minute rule still applies, but for 8KB pages instead of 1KB pages (due to a different technology and price ratio)
- “one-minute-sequential rule: [...] sequential operations should use main memory to cache data if the algorithm will revisit the data within a minute.”

[Gra 1997]

## practice

00110-1141/100706-100000

Article development led by C1008

summary

Revisiting Gray and Putzolu's famous rule in the age of Flash.

BY GOETZ GRAEFE

### The Five-Minute Rule 20 Years Later (and How Flash Memory Changes the Rules)

IN 1987, JIM Gray and Gianfranco Putzolu published their now-famous five-minute rule<sup>1</sup> for trading off memory and I/O capacity. Their calculation compares the cost of holding a record (or page) permanently in memory with the cost of performing disk I/O each time the record (or page) is accessed, using appropriate fractional prices of RAM chips and disk drives. The name of their rule refers to the break-even interval between accesses. If a record (or page) is accessed more often, it should be kept in memory; otherwise, it should remain on disk and be read when needed.

Based on then-current prices and performance characteristics of Tandem equipment, Gray and Putzolu found the price of RAM to hold a 1KB record

was about equal to the (fractional) price of a disk drive required to access such a record every 300 seconds, which they rounded to five minutes. The break-even interval is about inversely proportional to the record size. Gray and Putzolu reported one hour for 100-byte records and two minutes for 4KB pages.

The five-minute rule was reviewed and revised 10 years later<sup>2</sup>. Lots of prices and performance parameters had changed (for example, the price of RAM had tumbled from \$1,000 to \$15 per megabyte). Nevertheless, the break-even interval for 4KB pages was still around five minutes. The first goal of this article is to review the five-minute rule after another 10 years.

Of course, both previous articles acknowledged that prices and performance vary among technologies and devices at any point in time (RAM for mainframes versus micro-computers, SCSI versus IDE disks, and so on). Improved studies are needed to re-evaluate the appropriate formulas for their environments and equipment. The values used here (in Table 1) are meant to be typical for 2007 technologies rather than universally accurate.

In addition to quantitative changes in price and performance, qualitative changes already under way will affect the software and hardware architectures of servers and, in particular, database systems. Database software will change radically with the advent of new technologies: virtualization with hardware and software support, as well as higher utilization goals for physical machines, many-core processors and transactional memory supported both in programming environments and hardware<sup>3</sup>. Deployment in containers housing thousands of processes and many terabytes of data<sup>4</sup> and flash memory that fill the gap between traditional RAM and traditional storage disks.

Flash memory fills between traditional RAM and persistent mass storage based on rotating disks in terms of acquisition cost, access

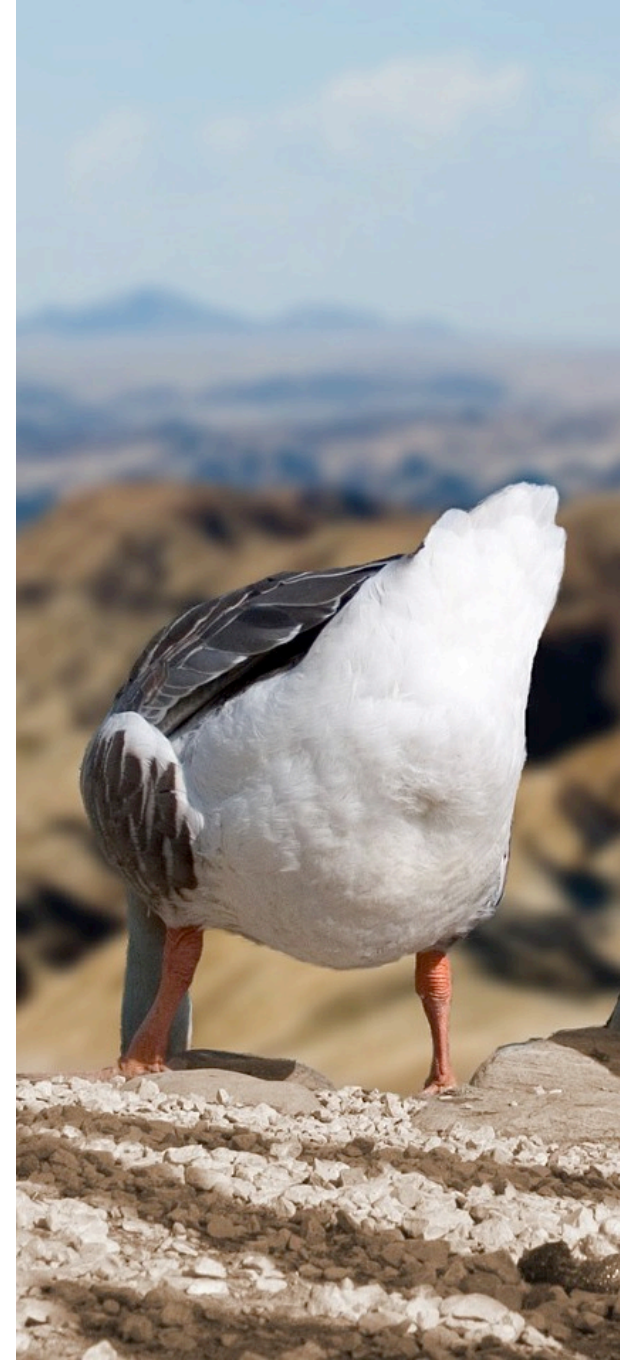
# The 5 minute rule 20 years later

- 5 minute rule still applies, but for 64KB pages instead of 8KB pages (due to a different technology and price ratio)
- or an alternative 5 minute rule differentiating RAM, Flash and SATA: RAM-Flash for 4KB pages / Flash-SATA for 256KB pages

[Gra 2009]

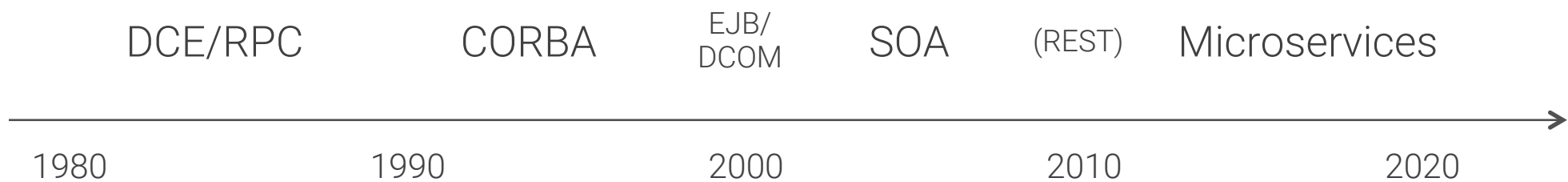
# Insights for today

- Validate your assumptions using models and data
- Heuristics can help to create options faster and better
- The best options often are not “all” or “nothing”
  - Balanced hybrid solutions often provide the highest value
- Re-validate your heuristics once in a while
- New technologies create completely new options & trade-offs



Distributed systems

# A short history of popular distributed system approaches



# Common pattern

1. Distributed systems are too complex for our developers
2. Let us hide the complexity behind some infrastructure
3. Provide interfaces that pretend local communication
4. Let the developers act as if they were implementing a local application
5. Let the infrastructure handle the complexities of distributed systems





Developer

“Everything will be fine!”



Promises deterministic behavior



Local facade & infrastructure

“Hold my beer!”



Delivers non-deterministic behavior



Distributed runtime



Will break the promise

## A Note on Distributed Computing

Jim Waldo  
Geoff Wyant  
Ann Wollrath  
Sam Kendall

SMLI TR-94-29


November 1994

### Abstract:

We argue that objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space. These differences are required because distributed systems require that the programmer be aware of latency, have a different model of memory access, and take into account issues of concurrency and partial failure.

We look at a number of distributed systems that have attempted to paper over the distinction between local and remote objects, and show that such systems fail to support basic requirements of robustness and reliability. These failures have been masked in the past by the small size of the distributed systems that have been built. In the enterprise-wide distributed systems foreseen in the near future, however, such a masking will be impossible.

We conclude by discussing what is required of both systems-level and application-level programmers and designers if one is to take distribution seriously.

 *Sun Microsystems*  
*Laboratories, Inc.*  
A Sun Microsystems, Inc. Business  
M/S 29-01  
2550 Garcia Avenue  
Mountain View, CA 94043

**email addresses:**  
jim.waldo@east.sun.com  
geoff.wyant@east.sun.com  
ann.wollrath@east.sun.com  
sam.kendall@east.sun.com

# A note on distributed computing

by Jim Waldo et al.

[Wal 1994]

**“Differences in latency, memory access, partial failure, and concurrency make merging of the computational models of local and distributed computing [...] unable to succeed.**

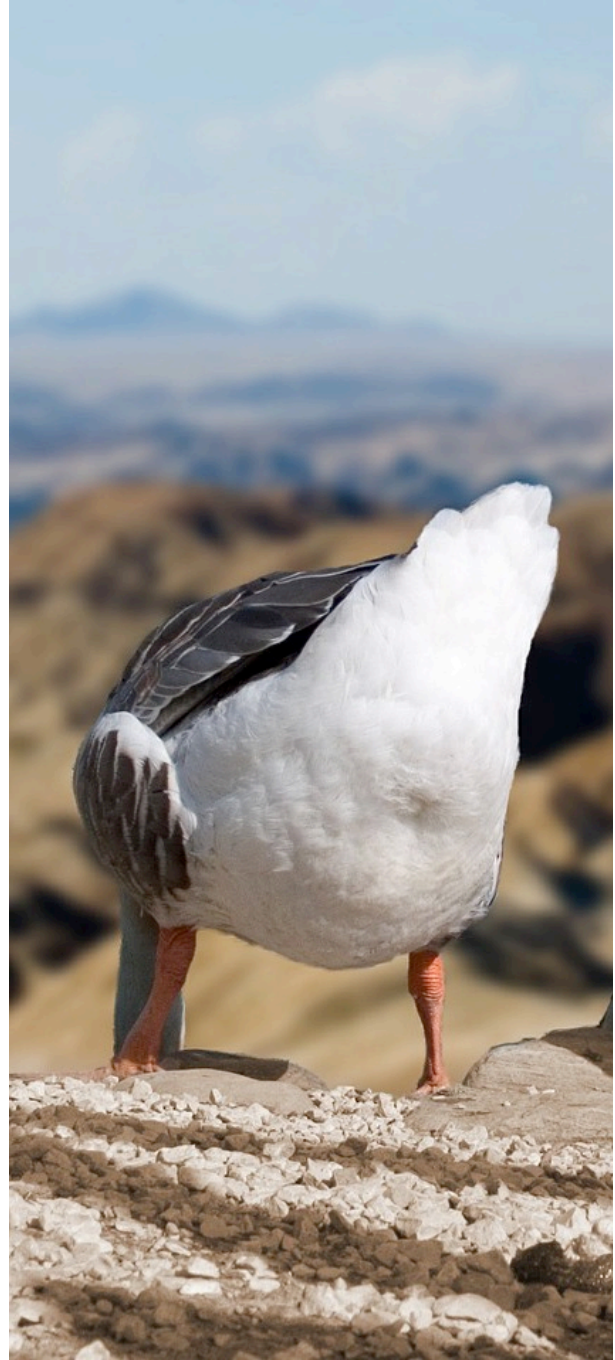
Merging the models by making local computing follow the model of distributed computing would [...] make local computing far more complex than is otherwise necessary.

Merging the models by attempting to make distributed computing follow the model of local computing requires ignoring the different failure modes and basic indeterminacy inherent in distributed computing, **leading to systems that are unreliable and incapable of scaling beyond small groups of machines** that are geographically co-located and centrally administered.”



# Insights for today

- **Distributed systems introduce non-determinism regarding**
  - Execution completeness
  - Message ordering
  - Communication timing
- You will be affected by this at the application level
  - Don't expect your infrastructure to hide all effects from you
  - Better have a plan to detect and recover from inconsistencies



## Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport  
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

**Key Words and Phrases:** distributed systems, computer networks, clock synchronization, multiprocess systems

**CR Categories:** 4.32, 5.29

### Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park CA 94025.  
© 1978 ACM 0001-0782/78/0700-0558 \$00.75

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

### The Partial Ordering

Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

# Time, clocks, and the ordering of events in a distributed system

by Leslie Lamport

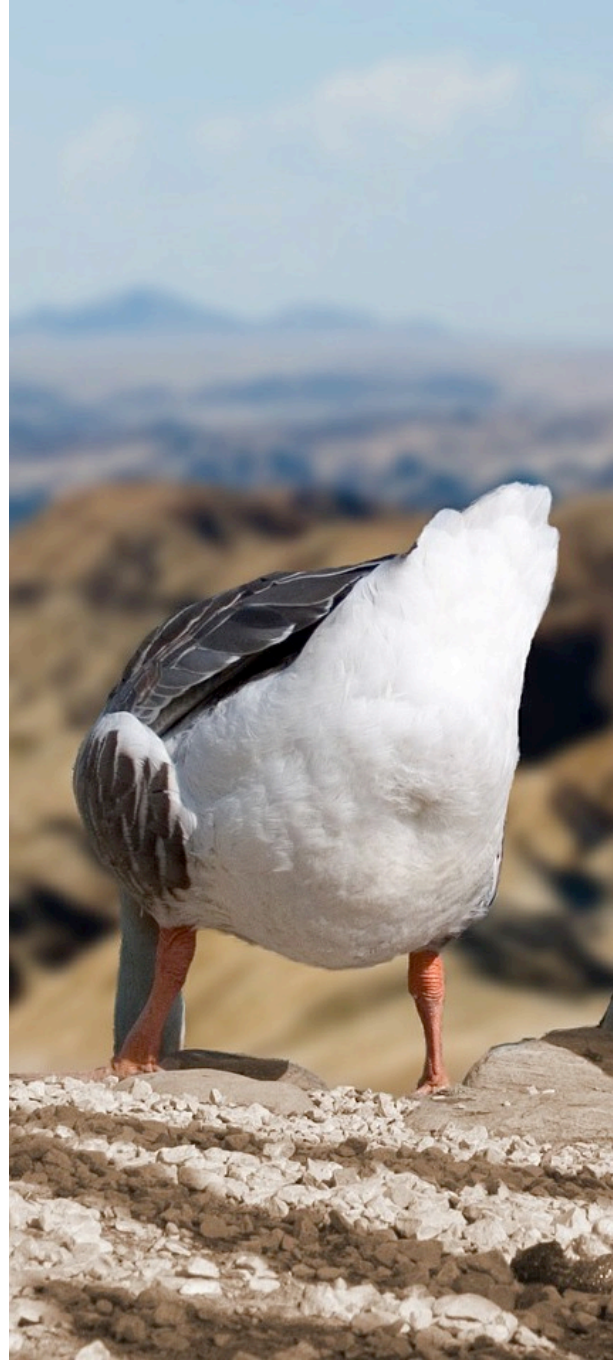
[Lam 1978]

**“In a distributed system, it is sometimes impossible to say that one of two events occurred first.** The relation “happened before” is therefore only a partial ordering of the events in the system. **We have found that problems often arise because people are not fully aware of this fact and its implications.”**

“However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. **Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time.”**

# Insights for today

- Do not rely on total ordering of events in your applications
  - Events can be concurrent
  - Messages can arrive out of order
- Do not rely on real clocks in distributed systems
  - Clock drift and skew can deceive you even in times of NTP
- Try to be independent of strict order and time



## Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

*Yale University, New Haven, Connecticut*

NANCY A. LYNCH

*Massachusetts Institute of Technology, Cambridge, Massachusetts*

AND

MICHAEL S. PATERSON

*University of Warwick, Coventry, England*

**Abstract.** The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

**Categories and Subject Descriptors:** C.2.2 [Computer-Communication Networks]: Network Protocols—*protocol architecture*; C.2.4 [Computer-Communication Networks]: Distributed Systems—*distributed applications; distributed databases; network operating systems*; C.4 [Performance of Systems]: Reliability, Availability, and Serviceability; F.1.2 [Computation by Abstract Devices]: Modes of Computation—*parallelism*; H.2.4 [Database Management]: Systems—*distributed systems; transaction processing*

**General Terms:** Algorithms, Reliability, Theory

**Additional Key Words and Phrases:** Agreement problem, asynchronous system, Byzantine Generals problem, commit problem, consensus problem, distributed computing, fault tolerance, impossibility proof, reliability

### 1. Introduction

The problem of reaching agreement among remote processes is one of the most fundamental problems in distributed computing and is at the core of many

Editing of this paper was performed by guest editor S. L. Graham. The Editor-in-Chief of JACM did not participate in the processing of the paper.

This work was supported in part by the Office of Naval Research under Contract N00014-82-K-0154, by the Office of Army Research under Contract DAAG29-79-C-0155, and by the National Science Foundation under Grants MCS-7924370 and MCS-8116678.

This work was originally presented at the 2nd ACM Symposium on Principles of Database Systems, March 1983.

Authors' present addresses: M. J. Fischer, Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520; N. A. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; M. S. Paterson, Department of Computer Science, University of Warwick, Coventry CV4 7AL, England

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1985 ACM 0004-5411/85/0400-0374 \$00.75

# Impossibility of distributed consensus with one faulty process

by Michael J. Fischer et al.

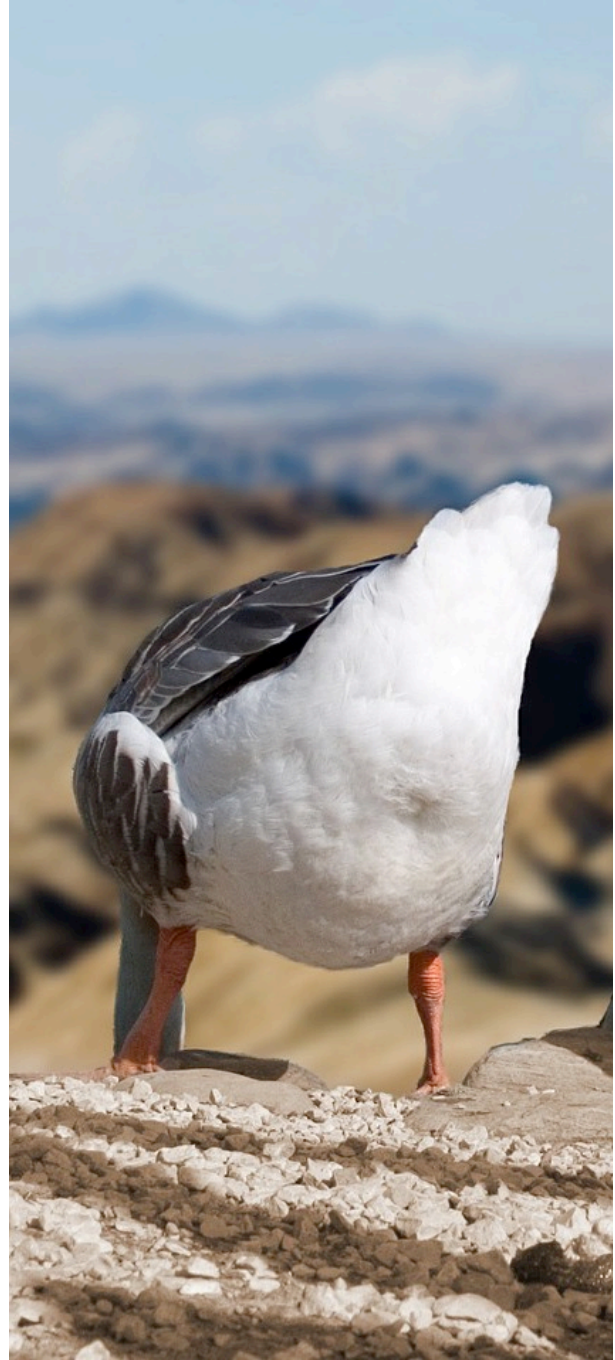
[Fis 1985]

“Reaching the type of agreement needed [...] is straightforward if the participating processes and the network are completely reliable. However, real systems are subject to a number of possible faults, such as process crashes, network partitioning, and lost, distorted, or duplicated messages.”

“We do not consider Byzantine failures, and we assume that the message system is reliable – it delivers all messages correctly and exactly once. Nevertheless, **even with these assumptions, the stopping of a single process at an inopportune time can cause any distributed commit protocol to fail to reach agreement.**”

# Insights for today

- Do not implicitly assume a reliable system
  - Crashes and partitioning happen
  - Messages can be lost, distorted or arrive multiple times
- Be aware that many problems are hard or insolvable
  - Don't think "That cannot be that hard" without proof
  - Especially consensus and consistency are tricky issues





# **Towards Robust Distributed Systems**

**Dr. Eric A. Brewer**  
Professor, UC Berkeley  
Co-Founder & Chief Scientist, Inktomi

PODC Keynote, July 19, 2000

Towards robust distributed systems

by Dr. Eric A. Brewer

[Bre 2000]



“Classic distributed systems [research] focus on the computation, not the data. This is wrong, computation is the easy part.”

“DBMS research is about ACID (mostly). But we forfeit ‘C’ and ‘I’ for availability, graceful degradation, and performance.

**This tradeoff is fundamental.**

BASE:

- **B**asically **A**vailable
- **S**oft-state
- **E**ventual consistency”

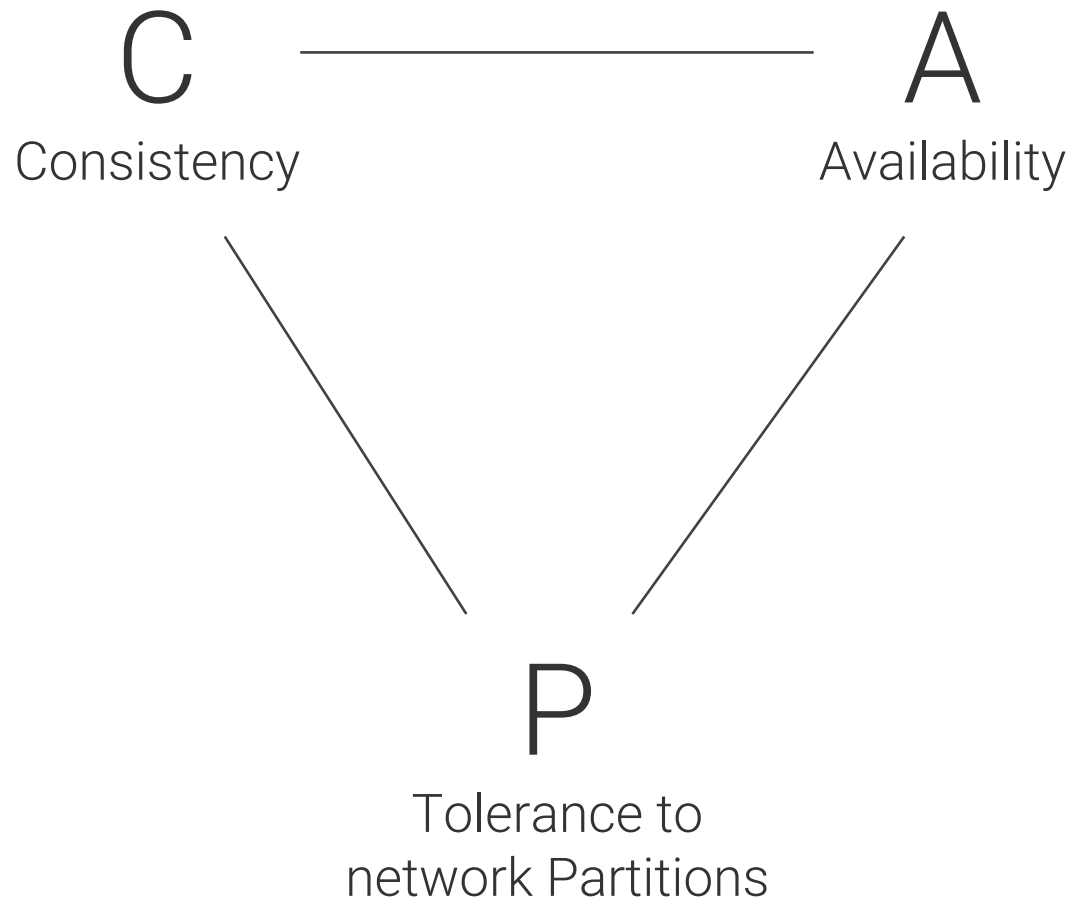
## ACID

- Strong consistency
- Isolation
- Focus on “commit”
- Nested transactions
- Availability?
- Conservative (pessimistic)
- Difficult evolution (e.g. schema)

## BASE

- Weak consistency (stale data OK)
- Availability first
- Best effort
- Approximate answers OK
- Aggressive (optimistic)
- Simpler!
- Faster
- Easier evolution

← **But I think it's a spectrum** →



Theorem:  
You can have **at most two**  
of these properties  
for any shared-data system

## Building on Quicksand

Pat Helland  
Microsoft Corporation  
One Microsoft Way  
Redmond, WA 98052 USA  
PHelland@Microsoft.com

Dave Campbell  
Microsoft Corporation  
One Microsoft Way  
Redmond, WA 98052 USA  
DavidC@Microsoft.com

### ABSTRACT

Reliable systems have always been built out of unreliable components [1]. Early on, the reliable components were small such as mirrored disks or ECC (Error Correcting Codes) in core memory. These systems were designed such that failures of these small components were transparent to the application. Later, the size of the unreliable components grew larger and semantic challenges crept into the application when failures occurred.

Fault tolerant algorithms comprise a set of idempotent sub-algorithms. Between these idempotent sub-algorithms, state is sent across the failure boundaries of the unreliable components. The failure of an unreliable component can then be tolerated as a takeover by a backup, which uses the last known state and drives forward with a retry of the idempotent sub-algorithm. Classically, this has been done in a linear fashion (i.e. one step at a time).

As the granularity of the unreliable component grows (from a mirrored disk to a system to a data center), the latency to communicate with a backup becomes unpalatable. This leads to a more relaxed model for fault tolerance. The primary system will acknowledge the work request and its actions *without* waiting to ensure that the backup is notified of the work. This improves the responsiveness of the system because the user is not delayed behind a slow interaction with the backup.

There are two implications of asynchronous state capture:

- 1) **Everything promised by the primary is probabilistic.** There is always a chance that an untimely failure shortly after the promise results in a backup proceeding without knowledge of the commitment. Hence, nothing is guaranteed!
- 2) **Applications must ensure eventual consistency [20].** Since work may be stuck in the primary after a failure and reappear later, the processing order for work cannot be guaranteed.

Platform designers are struggling to make this easier for their applications. Emerging patterns of eventual consistency and probabilistic execution may soon yield a way for applications to express requirements for a "looser" form of consistency while providing availability in the face of ever larger failures. As we will also point out in this paper, the patterns of probabilistic execution and eventual consistency are applicable to intermittently connected application patterns.

This paper recounts portions of the evolution of these trends, attempts to show the patterns that span these changes, and talks about future directions as we continue to "build on quicksand".

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIDR '09, In 7-10, 2009, Asilomar, Pacific Grove, CA USA  
Copyright 2009 ACM 1-58113-000-0/000004...\$5.00.

### Keywords

Fault Tolerance, Eventual Consistency, Reconciliation, Loose Coupling, Transactions

### 1. Introduction

There is an interesting connection between fault tolerance, offlineable systems, and the need for application-based eventual consistency. As we attempt to run our large scale applications spread across many systems, we cannot afford the latency to wait for a backup system to remain in synch with the system actually performing the work. This causes the server systems to look increasingly like offlineable client applications in that they do not know the authoritative truth. In turn, these server-based applications are designed to record their intentions and allow the work to interleave and flow across the replicas. In a properly designed application, this results in system behavior that is acceptable to the business while being resilient to an increasing number of system failures.

This paper starts by examining the concepts of fault tolerance and posits an abstraction for thinking about fault tolerant systems. Next, section 3 examines how fault tolerant systems have historically provided the ability to transparently survive failures without special application consideration by using synchronous checkpointing to send the application state to a backup. In section 4, we begin to examine what happens when we cannot afford the latency associated with the synchronous checkpointing of state to the backup and, instead, allow the checkpointing of state to be asynchronous. Section 5 examines in much more depth the ways in which an application must be modified to be true to its semantics while allowing asynchronous checkpointing of the application state to its backup. Section 6 looks at a couple of example applications which offer correct behavior while allowing delays (i.e. asynchrony) in checkpointing state to the backup. In section 7, we consider the management of resources when the operations may be reordered due to asynchrony. Section 8 examines the relationship between this class of eventual consistency and the CAP (Consistency, Availability, and Partition-tolerance) Theory. Finally, in section 9, we consider some areas for future work.

### 2. An Abstraction for Fault Tolerance

In section 2, we discuss the broad ideas required to build a fault tolerant system. First, we start by describing the external behavior of the systems we are considering. Next, we describe what it can mean for these systems to offer transparent fault tolerance and not require special application consideration to cope with failures. Then, we quickly consider the issues associated with scalability of these systems. Finally, we briefly discuss the role of transactions in the composition of these fault tolerant systems.

Building on quicksand  
by Pat Helland and Dave Campbell

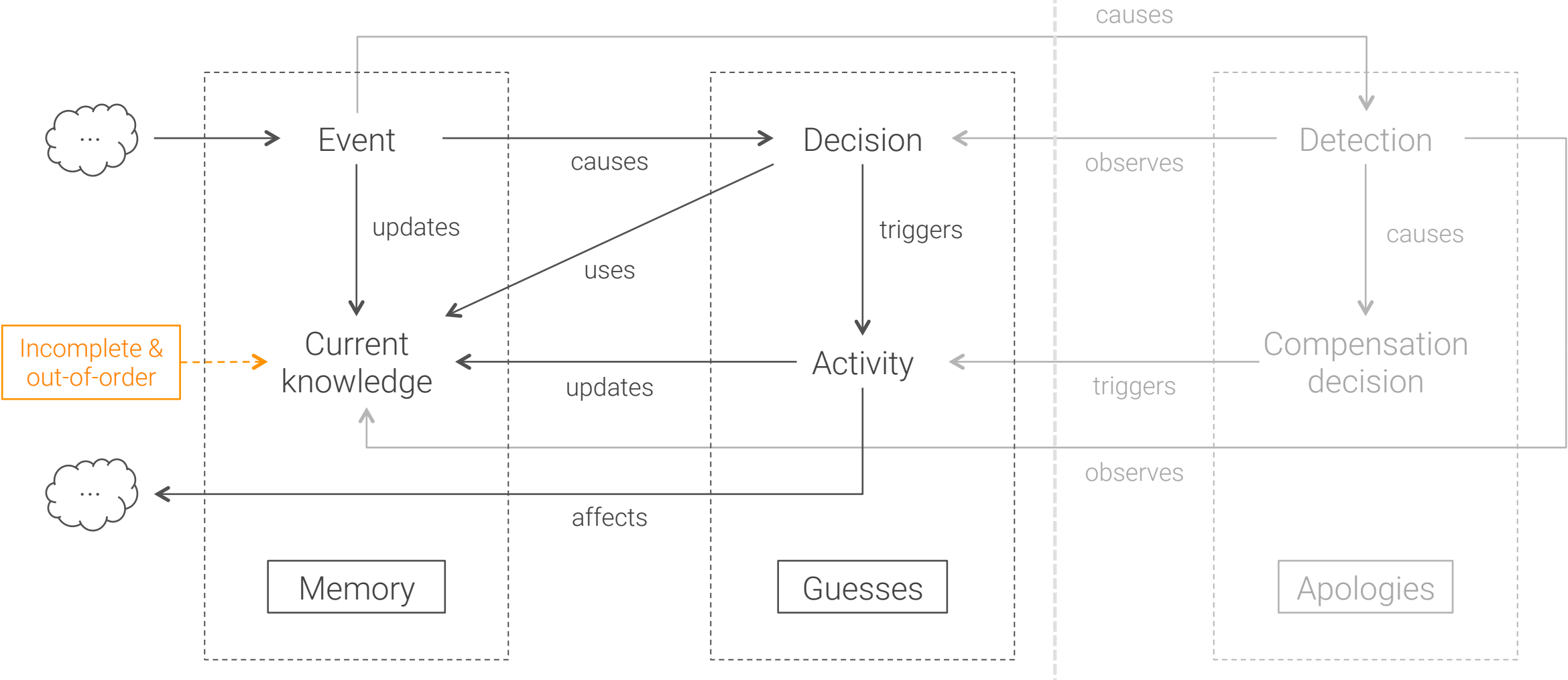
“Arguably, all computing really falls into three categories: **memories, guesses, and apologies.**

The idea is that everything is done locally with a subset of the global knowledge. You know what you know when an action is performed. **Since you have only a subset of the knowledge, your actions are really only guesses.**

When your knowledge as a replica increases, you may have an “Oh, crap!” moment. Reconciling your actions (as a replica) with the actions of an evil-twin of yours may result in recognition that there’s a mess to clean up. That may involve apologizing for your behavior (or the behavior of a replica).”

This part we usually implement assuming a perfect global knowledge in each node

This part we usually do not implement



“In a loosely coupled world choosing some level of availability over consistency, it is best to think of all computing as memories, guesses, and apologies.”

[Hel 2009]

# Wrapping up

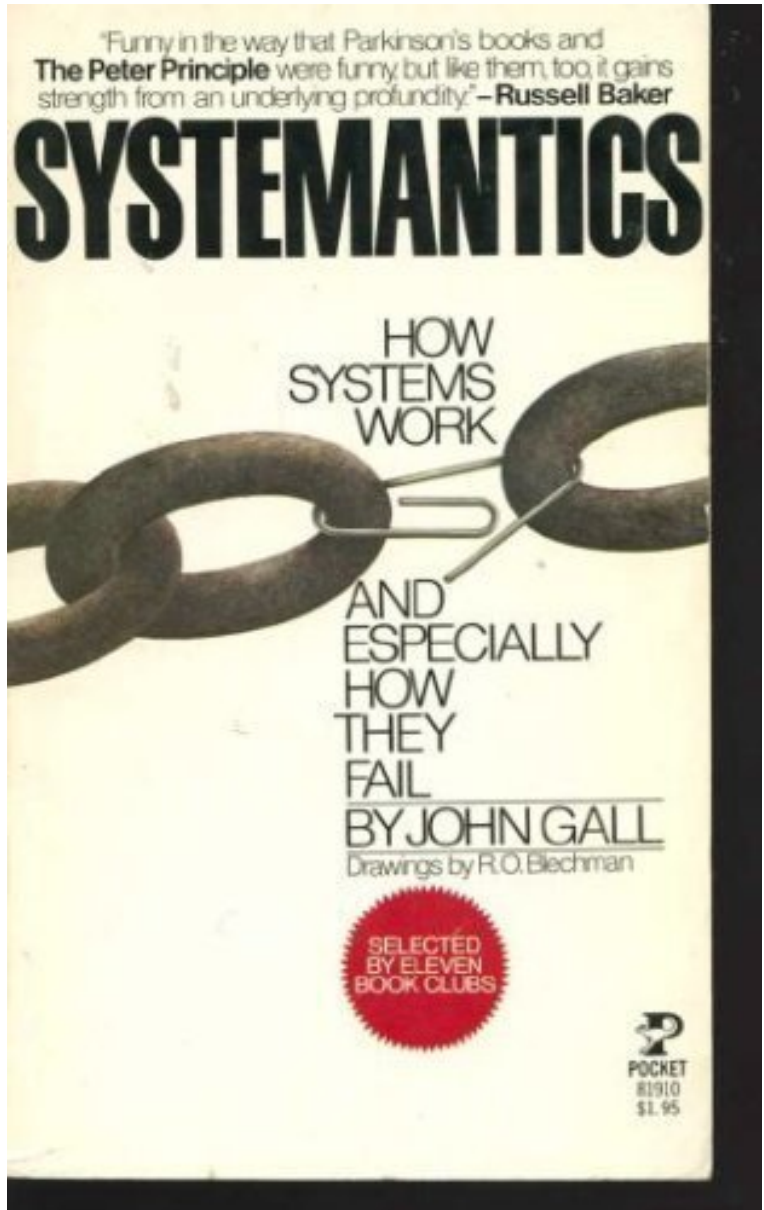
## Distributed systems

- Non-determinism of distributed systems changes everything
  - Traditional deterministic thinking not sufficient anymore
  - Effects of distribution cannot be hidden (or ignored)
- Simple problems can be hard in distributed environments
- Understand your options and trade-offs (really!)
- Think in concepts like memory, guesses and apologies





# Complexity and system design



## Systemantics

by John Gall

[Gal 1975]

“If anything can go wrong, it will.” (Murphy's law)

“**Complex systems exhibit unexpected behavior.**” (Generalized uncertainty principle)

“A large system, produced by expanding the dimensions of a smaller system, does not behave like the smaller system.” (Climax design theorem)

**“A complex system that works is invariably found to have evolved from a simple system that worked. (Working complex systems axiom)**

The parallel proposition also appears to be true:

**A complex system designed from scratch never works and cannot be made to work. You have to start over, beginning with a working simple system.”**

“Destiny is largely a set of unquestioned assumptions.”

[Gal 1975]

### How Complex Systems Fail

*(Being a Short Treatise on the Nature of Failure; How Failure is Evaluated; How Failure is Attributed to Proximate Cause; and the Resulting New Understanding of Patient Safety)*

Richard I. Cook, MD

Cognitive technologies Laboratory  
University of Chicago

**1) Complex systems are intrinsically hazardous systems.**

All of the interesting systems (e.g. transportation, healthcare, power generation) are inherently and unavoidably hazardous by the own nature. The frequency of hazard exposure can sometimes be changed but the processes involved in the system are themselves intrinsically and irreducibly hazardous. It is the presence of these hazards that drives the creation of defenses against hazard that characterize these systems.

**2) Complex systems are heavily and successfully defended against failure.**

The high consequences of failure lead over time to the construction of multiple layers of defense against failure. These defenses include obvious technical components (e.g. backup systems, 'safety' features of equipment) and human components (e.g. training, knowledge) but also a variety of organizational, institutional, and regulatory defenses (e.g. policies and procedures, certification, work rules, team training). The effect of these measures is to provide a series of shields that normally divert operations away from accidents.

**3) Catastrophe requires multiple failures – single point failures are not enough..**

The array of defenses works. System operations are generally successful. Overt catastrophic failure occurs when small, apparently innocuous failures join to create opportunity for a systemic accident. Each of these small failures is necessary to cause catastrophe but only the combination is sufficient to permit failure. Put another way, there are many more failure opportunities than overt system accidents. Most initial failure trajectories are blocked by designed system safety components. Trajectories that reach the operational level are mostly blocked, usually by practitioners.

**4) Complex systems contain changing mixtures of failures latent within them.**

The complexity of these systems makes it impossible for them to run without multiple flaws being present. Because these are individually insufficient to cause failure they are regarded as minor factors during operations. Eradication of all latent failures is limited primarily by economic cost but also because it is difficult before the fact to see how such failures might contribute to an accident. The failures change constantly because of changing technology, work organization, and efforts to eradicate failures.

**5) Complex systems run in degraded mode.**

A corollary to the preceding point is that complex systems run as broken systems. The system continues to function because it contains so many redundancies and because people can make it function, despite the presence of many flaws. After accident reviews nearly always note that the system has a history of prior 'proto-accidents' that nearly generated catastrophe. Arguments that these degraded conditions should have been recognized before the overt accident are usually predicated on naïve notions of system performance. System operations are dynamic, with components (organizational, human, technical) failing and being replaced continuously.

# How complex systems fail

by Richard I. Cook

[Coo 1998]

**“Catastrophe requires multiple failures – single point failures are not enough.**

[...] Overt catastrophic failure occurs when small, apparently innocuous failures join to create opportunity for a systemic accident. Each of these small failures is necessary to cause catastrophe but only the combination is sufficient to permit failure.”

**“Complex systems run in degraded mode.**

[...] complex systems run as broken systems. The system continues to function because it contains so many redundancies and because people can make it function, despite the presence of many flaws.”

**“Post-accident attribution accident to a ‘root cause’ is fundamentally wrong.** Because overt failure requires multiple faults, there is no isolated ‘cause’ of an accident. There are multiple contributors to accidents. [...] Indeed, it is the linking of these causes together that creates the circumstances required for the accident. Thus, no isolation of the ‘root cause’ of an accident is possible. The evaluations based on such reasoning as ‘root cause’ do not reflect a technical understanding of the nature of failure **but rather the social, cultural need to blame specific, localized forces or events for outcomes.**”

[Coo 1998]



**“Hindsight biases post-accident assessments of human performance.**

Knowledge of the outcome makes it seem that events leading to the outcome should have appeared more salient to practitioners at the time than was actually the case. **This means that ex post facto accident analysis of human performance is inaccurate.”**

**“Safety is a characteristic of systems and not of their components**

Safety is an emergent property of systems; it does not reside in a person, device or department of an organization or system. Safety cannot be purchased or manufactured; it is not a feature that is separate from the other components of the system.”

# No Silver Bullet

## Essence and Accidents of Software Engineering

Frederick P. Brooks, Jr.  
University of North Carolina at Chapel Hill

**Fashioning complex conceptual constructs is the essence; accidental tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.**

**O**f all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.

The familiar software project, at least as seen by the nontechnical manager, has something of this character; it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet—something to make software costs drop as rapidly as computer hardware costs do.

But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity. In this article, I shall try to show why, by examining both the nature of the software problem and the properties of the bullets proposed.

Skepticism is not pessimism, however. Although we see no startling break-

This article was first published in *Information Processing '86*, ISBN No. 0-444-70077-3, H.-J. Kugler, ed., Elsevier Science Publishers B.V. (North-Holland) © IFIP 1986.

throughs—and indeed, I believe such to be inconsistent with the nature of software—many encouraging innovations are under way. A disciplined, consistent effort to develop, propagate, and exploit these innovations should indeed yield an order-of-magnitude improvement. There is no royal road, but there is a road.

The first step toward the management of disease was replacement of demon theories and humours theories by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions. It told workers that progress would be made stepwise, at great effort, and that a persistent, unremitting care would have to be paid to a discipline of cleanliness. So it is with software engineering today.

### **Does it have to be hard?—Essential difficulties**

Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any—no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware.

No silver bullet

by Frederick P. Brooks, Jr.

“I divide [the difficulties of software technology] into **essence**, the difficulties inherent to the nature of software, and **accidents**, those difficulties that today attend its production but are not inherent.”

“Let us consider the inherent properties of this irreducible essence of modern software systems: **complexity, conformity, changeability and invisibility.**”

## Out of the Tar Pit

Ben Moseley                      Peter Marks  
ben@moseley.name              public@indigomail.net

February 6, 2006

### Abstract

Complexity is the single major difficulty in the successful development of large-scale software systems. Following Brooks we distinguish *accidental* from *essential* difficulty, but disagree with his premise that most complexity remaining in contemporary systems is essential. We identify common causes of complexity and discuss general approaches which can be taken to eliminate them where they are accidental in nature. To make things more concrete we then give an outline for a potential complexity-minimizing approach based on *functional programming* and *Codd's relational model of data*.

### 1 Introduction

The “software crisis” was first identified in 1968 [NR69, p70] and in the intervening decades has deepened rather than abated. The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand. We believe that the major contributor to this complexity in many systems is the handling of *state* and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are *code volume*, and explicit concern with the *flow of control* through the system.

The classical ways to approach the difficulty of state include object-oriented programming which tightly couples state together with related behaviour, and functional programming which — in its pure form — eschews state and side-effects all together. These approaches each suffer from various (and differing) problems when applied to traditional large-scale systems.

We argue that it is possible to take useful ideas from both and that — when combined with some ideas from the relational database world —

# Out of the tar pit

by Ben Moseley and Peter Marks

[Mos 2006]

**“Complexity is the root cause of the vast majority of problems with software today.** Unreliability, late delivery, lack of security – often even poor performance in large-scale systems can all be seen as deriving ultimately from unmanageable complexity. The primary status of complexity as the major cause of these other problems comes simply from the fact that **being able to understand a system is a prerequisite** for avoiding all of them, and of course it is this which complexity destroys.”

“[...] it is our belief that the single biggest remaining cause of complexity in most contemporary large systems is state, **and the more we can do to limit and manage state, the better.**

Control is basically about the order in which things happen. [...] **Most traditional programming languages do force a concern with ordering** [...] The difficulty is that when control is an implicit part of the language [...], then every single piece of program must be understood in that context [...]

The final cause of complexity that we want to examine in any detail is sheer code volume. [...] **in most systems complexity definitely does exhibit nonlinear increase with size** (of the code). This non-linearity in turn means that it's vital to reduce the amount of code to an absolute minimum.”

“Finally there are other causes [...] All of these other causes come down to the following three (inter-related) principles:

- **Complexity breeds complexity:** [...] This covers all complexity introduced as a result of not being able to clearly understand a system. [...] This is particularly true in the presence of time pressures.
- Simplicity is Hard: [...] **Simplicity can only be attained if it is recognized, sought and prized.**
- Power corrupts: [...] in the absence of language-enforced guarantees mistakes (and abuses) will happen. [...] The bottom line is that **the more powerful a language** (i.e. the more that is possible within the language), **the harder it is to understand systems constructed in it.**”

“Hence we define the following two types of complexity:

- Essential Complexity is inherent in, and **the essence of, the problem (as seen by the users)**.
- Accidental Complexity is all the rest — **complexity with which the development team would not have to deal in the ideal world** (e.g. complexity arising from performance issues and from suboptimal language and infrastructure).

When it comes to accidental and essential complexity we firmly believe that the former exists and that the goal of software engineering must be both to eliminate as much of it as possible, and to assist with the latter.”



# Wrapping up

## Complexity and system design

- Complexity is hard
  - Hard to understand, exhibits unexpected behavior
  - Cannot be designed from scratch, but must evolve
  - Exhibits complex failure modes
- Distinguish essential and accidental complexity
  - Essential complexity is needed to solve the problem
  - Accidental complexity is everything else – try hard to avoid it



Closing thought

## HOW TO USE CONSCIOUS PURPOSE WITHOUT WRECKING EVERYTHING

By John Gall, MS, MD, FAAP

A talk prepared for presentation at the annual Gilbfest, London, UK, June 25, 2012

### CONTENTS

#### I. INTRODUCTION

1. POWERPOINT SLIDES
2. GOETHE'S *FAUST*

#### II. SYSTEMANTICS SINCE 1976

1. BLACK SWANS AND OTHER *BÊTES NOIRES*
2. SYSTEMS IDEAS SINCE 1976

#### III. FEEDBACK, FLEXIBILITY, CREATIVITY

##### 1. FEEDBACK

**Stafford Beer and feedback in living organisms**  
**Mirror Neurons**  
**Long-term memory**  
**Integration of experience**

##### 2. FLEXIBILITY

**Catherine the Great of Russia**

##### 3. CREATIVITY

#### IV. PROBLEM SOLVING AS A PROBLEM TO SOLVE

##### 1. MOTHER NATURE

**Dinosaurs into Birds**

##### 2. PROBLEM SOLVING AT THE FIRST LEVEL: VARIETY

##### 3. PROBLEM-SOLVING AT THE SECOND LEVEL: NOVELTY

##### 4. PROBLEM-SOLVING AT THE THIRD LEVEL: HIGHER-LEVEL CREATIVITY

**Moving up to the next level of recursion**  
**Dolphins**

##### 5. THE SCHOOL SYSTEM

**The deadly command to Pay Attention**  
**Left brain/right brain**  
**Daydreaming and creativity**  
**Command-and-Control**  
**Dancing is a stochastic process**

#### V. LEVELS OF RECURSION. SYSTEMS IN SYSTEMS OF SYSTEMS

1. WHEN IS AN APPLE NOT AN APPLE?
2. POSIWID

#### VI. AFTER SYSTEMANTICS, WHAT?

1. THE BOUNDARY PROBLEM
2. CONNECTEDNESS
3. WHOLENESS

How to use conscious purpose  
without wrecking everything

by John Gall

[Gal 2012]

“I think that in this one play [Faust] Goethe has posed a core problem for modern man — namely: **being in possession of the powerful tool of conscious thought, how to use it without wrecking everything.**”

[Gal 2012]

The Humble Programmer.

by

Edsger W. Dijkstra

As a result of a long sequence of coincidences I entered the programming profession officially on the first spring morning of 1952 and as far as I have been able to trace, I was the first Dutchman to do so in my country. In retrospect the most amazing thing was the slowness with which, at least in my part of the world, the programming profession emerged, a slowness which is now hard to believe. But I am grateful for two vivid recollections from that period that establish that slowness beyond any doubt.

After having programmed for some three years, I had a discussion with A. van Wijngaarden, who was then my boss at the Mathematical Centre in Amsterdam, a discussion for which I shall remain grateful to him as long as I live. The point was that I was supposed to study theoretical physics at the University of Leiden simultaneously, and as I found the two activities harder and harder to combine, I had to make up my mind, either to stop programming and become a real, respectable theoretical physicist, or to carry my study of physics to a formal completion only, with a minimum of effort, and to become....., yes what? A programmer? But was that a respectable profession? For after all, what was programming? Where was the sound body of knowledge that could support it as an intellectually respectable discipline? I remember quite vividly how I envied my hardware colleagues, who, when asked about their professional competence, could at least point out that they knew everything about vacuum tubes, amplifiers and the rest, whereas I felt that, when faced with that question, I would stand empty-handed. Full of misgivings I knocked on van Wijngaarden's office door, asking him whether I could "speak to him for a moment"; when I left his office a number of hours later, I was another person. For after having listened to my problems patiently, he agreed that up till that moment there was not much of a programming discipline, but then he went on to explain quietly that automatic computers were here to stay, that we were just at the beginning and could not I be one of the persons called to make programming a respectable discipline in the years to come? This was a turning point in my life and I completed my study of physics formally as quickly as I could. One moral of the above story is, of course, that we must be very careful when we give advice to younger people: sometimes they follow it!

# The humble programmer

by Edsger W. Dijkstra

[Dij 1972]

“We shall do a much better programming job, provided that **we will approach the task with a full appreciation of its tremendous difficulty**, provided that we stick to modest and elegant programming languages, provided that **we respect the intrinsic limitation of the human mind and approach the tasks as Very Humble Programmers.**”

software engineering



“We shall do a much better ~~programming~~ job, provided that **we will approach the task with a full appreciation of its tremendous difficulty**, provided that we stick to modest and ~~elegant programming languages~~, provided that **we respect the intrinsic limitation of the human mind and approach the tasks as Very Humble Programmers.**”



Software Engineers

fit-for-purpose tools



Resources



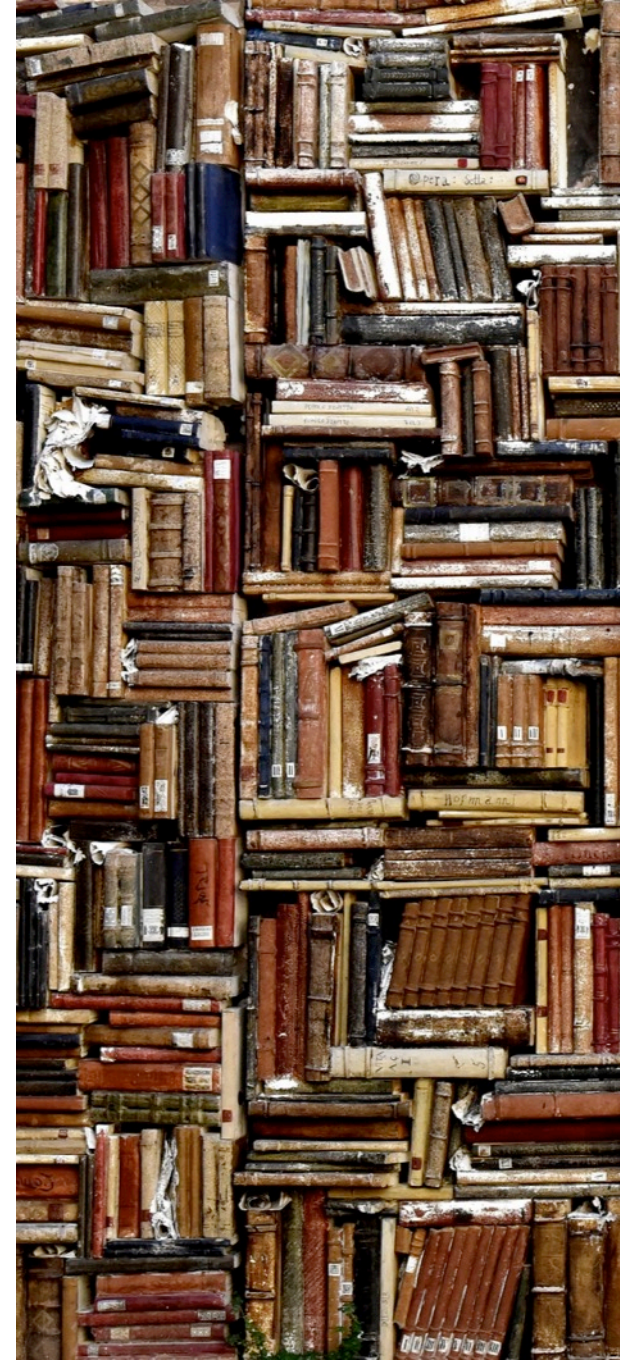
# References (1/5)

[Bre 2000] Eric A. Brewer, "Towards robust distributed systems", Keynote, Symposium on Principles of Distributed Computing (PODC) 2000

[Bro 1986] Frederick P. Brooks, jr., "No silver bullet", from H.-J. Kugler (ed.), "Information Processing 1986: World Congress Proceedings (IFIP congress series, vol 10)", 1986

[Bro 1995] Frederick P. Brooks, jr., "The tar pit", from "The mythical man-month", anniversary edition 1995

[Coo 1998] Richard I. Cook, "How complex systems fail", Cognitive technologies Laboratory, University of Chicago, 1998



# References (2/5)

[Dij 1968] Edsger W. Dijkstra, "Go to statement considered harmful", Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148

[Dij 1972] Edsger W. Dijkstra, "The humble programmer", Communications of the ACM, Vol. 15, No. 10, October 1972, pp. 859–866

[Fis 1985] Michael J. Fischer, Nancy A. Lynch, Michael S. Paterson, "Impossibility of distributed consensus with one faulty process", Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985, pp. 374-382

[Gal 1975] John Gall, "Systemantics", 1975



# References (3/5)

[Gal 2012] John Gall, "How to use conscious purpose without wrecking everything", Gilbfest, London, UK, June 25, 2012

[Gra 1986] Jim Gray, Franco Putzolu, "The 5 minute rule for trading memory for disc accesses and the 5 byte rule for trading memory for CPU time", Tandem Computer Technical Report 86.1, 1986

[Gra 1997] Jim Gray, Goetz Graefe, "The five-minute rule ten years later, and other computer storage rules of thumb", Microsoft Research Technical Report MSR-TR-97-33, 1997

[Gra 2009] Goetz Graefe, "The five-minute rule 20 years later (and how flash memory changes the rules)", Communications of the ACM, Vol. 52, No. 7, July 2009, pp. 48-59



# References (4/5)

[Hel 2009] Pat Helland, Dave Campbell, "Building on quicksand", Conference on Innovative Data Systems Research (CIDR) 2009

[Hoa 1981] Sir Charles Anthony Richard Hoare, "The emperor's old clothes", Communications of the ACM, Vol. 24, No. 2, February 1981, pp. 75-83

[Lam 78] Leslie Lamport, "Time, clocks, and the ordering of events in distributed systems", Communications of the ACM, Vol. 21, No. 7, July 1978, pp. 558-565

[Lam 2006] Leslie Lamport, "Computation and state machines", 2006



# References (5/5)

[Mos 2006] Ben Mosely, Peter Marks, "Out of the tar pit", 2006

[Par 1971] David L. Parnas, "Information distribution aspects of design methodology", Carnegie Mellon University, Research Showcase @CMU, 1971

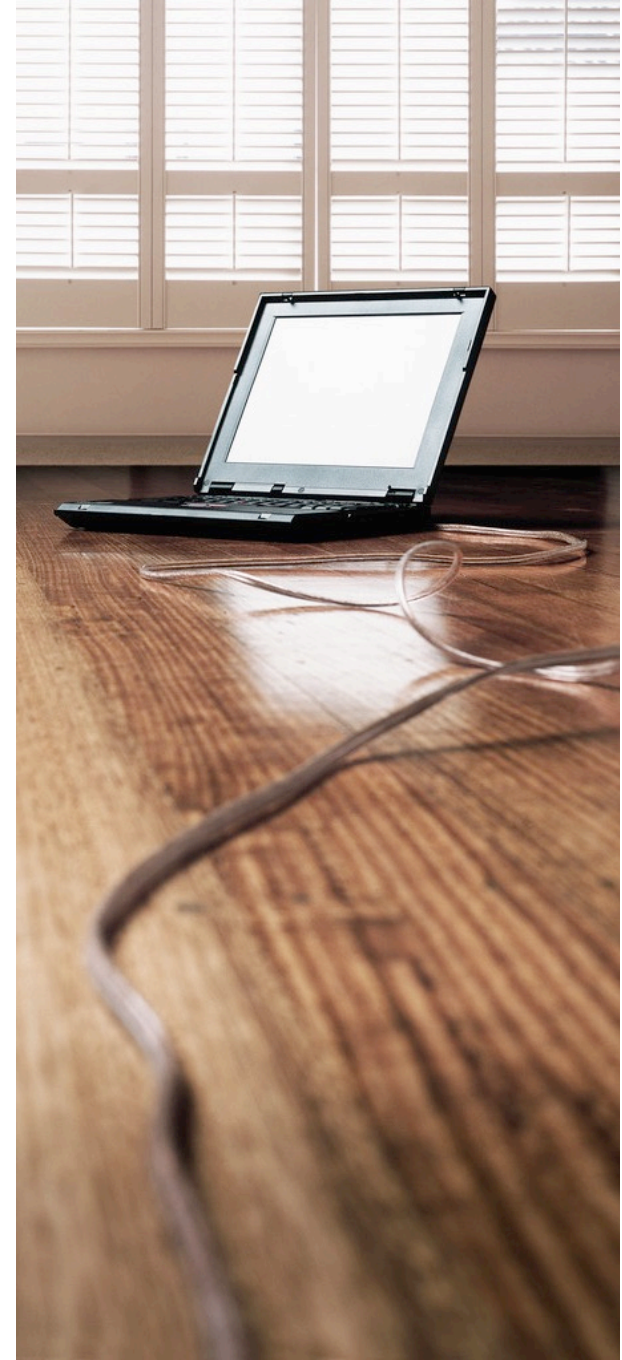
[Par 1972] David L. Parnas, "On the criteria to be used in decomposing systems into modules", Communications of the ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058

[Wal 1994] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall, "A note on distributed computing", Sun Microsystems Laboratories, Inc. TR-94-29, 1994



# More papers to discover

- **Papers we love** (<http://paperswelove.org/>)
- **The morning paper** (<https://blog.acolyer.org/>)
- **Edsger W. Dijkstra Prize in Distributed Computing** ([https://en.wikipedia.org/wiki/Dijkstra\\_Prize](https://en.wikipedia.org/wiki/Dijkstra_Prize))



# Uwe Friedrichsen

IT traveller.

Connecting the dots.

Attracted by uncharted territory.

CTO at codecentric.

<https://www.slideshare.net/ufried>

<https://medium.com/@ufried>



@ufried