Hints and Principles for Computer System Design

Butler Lampson
September 12, 2019

Abstract

This new short version of my 1983 paper suggests the goals you might have for your system—Simple, Timely, Efficient, Adaptable, Dependable, Yummy (STEADY)—and effective techniques for achieving them—Approximate, Incremental, Divide & Conquer (AID). It gives a few principles for system design that are more than just hints, and many examples of how to apply the hints and principles.

1 Introduction

There are three rules for writing a novel. Unfortunately, no one knows what they are. —Somerset Maugham[Q31]

You got to be careful if you don't know where you're going, because you might not get there. —Yogi Berra[Q4]

In 1983 I wrote a paper on "Hints for Computer System Design" for the Symposium on Operating System Principles.[R29] I reread that paper every two or three years, and for more than 15 years I saw no reason to rewrite or extend it; I had written what I knew about personal distributed computing, operating systems, languages, networking, databases, and fault tolerance, and computer systems were continuing the work of the 1970s on these things. But since the mid-1990s the Internet, mobile phones, the World Wide Web, search engines, social media, electronic commerce, malware, phishing, robots and the Internet of Things have become part of the fabric of everyday life, and concurrency and scaling are now dominant themes in systems. So for the last few years I've been trying to write a new version.

Then I could fit nearly everything I knew into a reasonable number of pages, but today computing is much more diverse and I know a lot more; this paper is unreasonably long. I couldn't find a single way to organize it, so I've taken several different perspectives and put in links (like this) to help you find what you need, especially if you read it online. There's also a set of principles (based on the idea of abstraction) that almost always apply, and a collection of oppositions (simple vs. rich, declarative vs. imperative, etc.) that suggest different ways to look at things.

The hints themselves are organized along three axes, corresponding to three time-honored questions, with a catchy summary: STEADY with AID by ART.

What?	**Goals**	STEADY	—**S**imple, **T**imely, **E**fficient, **A**daptable, **D**ependable, **Y**ummy
How?	**Techniques**	with AID	—**A**pproximate, **I**ncremental, **D**ivide & Conquer
When, who?	**Process**	by ART	—**A**rchitecture, **A**utomate, **R**eview, **T**echniques, **T**est

These are just hints. They are not
- novel (with a few exceptions),
- foolproof recipes, guaranteed to work,
- precisely formulated laws of system design or operation,
- consistent,
- always appropriate, or

approved by all the leading experts.

Skip over the ones you find wrong, useless or boring.

The paper begins with the importance of a point of view and a list of the oppositions, which can help you decide on priorities and structure for a system. §2 presents the principles: abstraction, specs, code and modularity. In §3 each goal gets a section on the techniques that support it, followed by one for techniques that didn't fit under a goal. "Efficient" gets by far the most space here, followed by "dependable"; this is because locality and concurrency fall naturally under the first and redundancy under the second, and these three are fundamental to today's systems. Finally there's a short nontechnical section §4 on process, and a discussion of each opposition in §5. Throughout, short slogans highlight the most important points without any nuance, and quotations give a sometimes cynical commentary on the text.

There are lots of examples to illustrate specific points; I've tried to choose well-known ones, but you may have to look them up to see the point. I've also told some longer stories, marked with ». Many things fit in more than one place, so there are many cross-reference links. A term of art is in italics the first time it's used; it's a good starting point for a web search.

This is not a review article; the work I cite is the work I know about, not necessarily the earliest or the best. I've given some references to material that expands on the ideas or examples, but usually only when it would be hard to find with a web search.

There's a longer version of the paper here.

1.1 Goals, techniques and process

1.1.1 *Goals*—STEADY

[Data is not information,] Information is not knowledge, Knowledge is not wisdom, Wisdom is not truth, Truth is not beauty, Beauty is not love, Love is not music and Music is THE BEST — Frank Zappa[Q55]

By goals I mean general properties that you want your system to have, not the problem it tries to solve. You should want your system to be STEADY: **S**imple, **T**imely, **E**fficient, **A**daptable, **D**ependable, and **Y**ummy. Since you can't have all these good thing at the same time, you need to decide which goals are most important to you; engineering is about trade-offs.

Simple should always be the leading goal, and abstraction is the best tool for making things simpler, but neither one is a panacea. There's no substitute for getting it right. Three other goals are much more important now than in the 1980s: Timely, Adaptable, and Yummy.

- **Timely** (early in time to market) because cheap computer hardware means that both enterprises and consumers use computer systems in every aspect of daily life, and you can deploy a system as soon as the software is ready. If you can't deliver the system quickly, your competitor can.
- **Adaptable** because the Internet means that a system can go from having a few dozen users to having a few million in a few weeks. Also, user needs can change quickly, and for many applications it's much more important to be agile than to be correct.
- **Yummy**[Q43] because many systems are built to serve consumers, who are much less willing than organizations to work hard to learn a system, and much more interested in fashions, features and fads. Even for professionals, the web, social media and GitHub mean that it's easy for enthusiasm to build up in defiance of formal procurement processes.

Goals	**Simple**	**Timely**	**Efficient**	**Adaptable**	**Dependable**	**Yummy**
As questions	Is it clean?	Is it ready?	Is it fast?	Can it evolve?	Does it work?	Will it sell?
As nouns	Simplicity	Time to market	Cost	Adaptability	Dependability	Features
Alliterative	Frugal	First	Fast	Flexible	Faithful	Fine/Fancy

1.1.2 *Techniques*—AI^{n}D

Techniques are the ideas and tools that you use to build a system; knowing about them keeps you from reinventing the wheel. The most important ones are about abstraction and specs; those are principles, not just hints. Most of the rest fall under three major headings:

- **Approximate** rather than exact, perfect or optimal results are usually good enough, and often much easier and cheaper to achieve. Loose rather than tight specs are more likely to be satisfied, especially when there are failures or changes. Lazy or speculative execution helps to match resources with needs.
- **Incremental** design has several aspects, many beginning with "i". The most important is to build the system out of *i*ndependent, *i*solated parts with *i*nterfaces that you can put together in different ways. Such parts are easier to get right, evolve and secure, and with *i*ndirection and virtualization you can reuse them in many different environments. *I*terating the design rather than deciding everything up front keeps you from getting too far out of touch with customers, and extensibility makes it easy for the system to evolve.
- **Divide and conquer** is the most important technique, especially abstractions with clean specs for organizing your system. This is the only way to maintain control when the system gets too big for one person's head, or when you come back to it later. Other aspects: making your system concurrent to exploit your hardware, redundant to handle failures, and recursive to reuse your work. The incremental techniques are an aspect of divide and conquer.

For each technique, many examples show how it's used and emphasize how widely applicable it is. A small number of ideas show up again and again, often concealed by the fact that people use different words for the same thing. The catalog below is both short and surprisingly complete.

Here are links to important techniques, to inspire you when you have a design problem.

Simple: abstraction, action, extensible, interface, predictable, relation, spec.
Efficient: algorithm, batch, cache, concurrent, lazy, local, shard, stream, summarize, translate.
Adaptable: dynamic, index, indirect, scale, virtualize.
Dependable: atomic, consensus, eventual, redundant, replicate, retry.
Incremental: becoming, indirect, interface, recursive, tree.

1.1.3 *Process*—ART

Process is who does what when, the mechanics of how you build and deploy a system: design, coding, testing, deployment, operations. The acronym is ART: **A**rchitecture, **A**utomation, **R**eview, **T**echniques, **T**esting. I know a lot less about this, since I've never been a manager, but people who've done it well have similar stories.

1.2 Points of view

A point of view is worth 80 points of IQ —Alan Kay[Q24]

A good way of thinking about a system makes things easier, just as the center-of-mass coordinate system makes dynamics problems easier. It's not that one viewpoint is more correct than another,

but that it's more convenient for some purpose. Many of the oppositions below reflect this idea. Here are some examples of alternative points of view, discussed in more detail later:

- Being vs. becoming: the state is the variable values (a map), or the actions that made it (a log).
- An interface adapter is part of a component or part of the environment.
- Iterative vs. recursive: do the same thing or divide into sub-cases until it's really simple.
- Declarative vs. imperative: a result is defined by its properties or by the steps that achieve it.
- Interpreter vs. compiler: different primitives get you different speed, size, or ease of change.

1.2.1 *Notation*

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. —Whitehead[Q53]

Notation is closely related to viewpoint, making something that's important easier to think about. Every system has at least some of its own notation: the datatypes and operations it defines, which are a domain-specific language (DSL) without its own syntax. More broadly, a notation can be general-purpose: a programming language like C or Python, or a library like the C++ standard template library. Or it can be specialized: a DSL like the Unix shell (for sequential string processing) or Julia (for numerical computation), or a library like TensorFlow (for machine learning).

A notation consists of:

- *Vocabulary* for naming relevant objects and actions (`grep`, `awk`, `cat`, etc. for the shell). Generic terms make it easier for people: "sort" for different sorting methods, "tree" for partially ordered or recursive structures. In a spec, the foundation should be mathematics, most often relations.
- *Syntax* for stringing them together (in the shell, "`|`" for pipes, "`>`" for redirect, etc.). In a DSL, syntax is a way to make common things in the domain easy to write and read. By contrast, a library for a general-purpose language has to live with the syntax of that language, typically method selection and function call.

1.3 Oppositions and slogans

I've looked at life from both sides now. —Joni Mitchell[Q33]

It often helps to think about design in terms of the opposition between two (or three) extremes. Here are some important ones, each with a few slogans that when properly interpreted reveal its (sometimes contradictory) essence. They are ordered by the first goal or technique they serve, with other goals in [brackets]. At the end of the paper there's a discussion of each one.

Goal	Opposition		Slogan
Principles	Spec ↔ code	[S]	Write a spec. Get it right. Keep it clean. Don't hide power. Leave it to the client.
Simple	Simple ↔ rich, fine ↔ features, general ↔ specialized	[Y]	KISS: Keep It Simple, Stupid. Do one thing well. Don't generalize. Don't hide power. Leave it to the client. Make it fast. Use brute force.
	Spec ↔ code	[P]	Keep secrets. Free the implementer. Good fences make good neighbors. Embrace nondeterminism. Abstractions are leaky.
	Perfect ↔ adequate, exact ↔ tolerant	[TD]	Just good enough. Flaky, springy parts.
	Declarative ↔ functional ↔ imperative	[E]	Say what you want. Make it atomic.

Timely	Precise ↔ approximate software	[D]	Get it right. Make it cool. Shipping is a feature.
Efficient			ABCs. Latency vs. bandwidth. Use theory. S^3: shard, stream or struggle. Make it atomic.
	Dynamic ↔ static	[A]	Stay loose. Pin it down. Shed load. Split resources.
	Indirect ↔ inline	[I]	Take a detour, see the world.
	Lazy ↔ eager ↔ speculative		Put it off. Take a flyer.
	Centralized ↔ distributed, share ↔ copy	[D]	Do it again. Do it twice. Find consensus.
Adaptable	Fixed ↔ evolving, monolithic ↔ extensible	[I]	The only constant is change. Make it extensible. Flaky, springy parts.
	Policy ↔ mechanism		It's OK to change your mind.
Dependable	Consistent ↔ available ↔ partitionable		Safety first. Always ready. Good enough.
	Generate ↔ check		Trust but verify.
Incremental	Being ↔ becoming		How did we get here? Don't copy, share.
	Iterative ↔ recursive, array ↔ tree		Treat the part like the whole.
Process			Build on a platform. Keep interfaces stable.

2 Principles

The ideas in this section are not just hints, they are the basic mental tools for system design.

2.1 **Abstraction**—Write a spec

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise. —Edsger Dijkstra[Q14]

Without a specification, a system cannot be wrong, it can only be surprising. —Gary McGraw[Q32]

If you're not writing a program, don't use a programming language. —Leslie Lamport[Q28]

Abstraction is the most important idea in computing. It's the way to make things simple enough that your limited brain can get the machine to do what you want, even though the details of what it does are too complicated for you to track: many, many steps and many, many bits of data. The idea is to have a *specification* for the computer system that tells you

- *what*: everything you need to know to use the system,
- but not *how*: anything about how it works internally, which this paper calls the *code*.

The spec describes the abstract *state* of the system (the values of its variables) using basic notions from mathematics, usually relations and their special cases: sets, sequences, tuples, functions, and graphs. For example, a file system spec describes a file as a pair: a size plus an array of that many bytes. Internally the code has data blocks, index blocks, buffer caches, storage allocators, crash recovery, etc., but none of this appears in the spec. The spec *hides* the complexity of the code from the client. Almost always the spec is much simpler, so the client's life is much easier.

The spec also describes the *actions* that read and change the state; a file has read, write, and set-length actions. An action a is just a set of possible transitions or *steps* from a pre-state s to a post-state s', so it too can be described by a relation, a predicate $a(s, s')$ on states that is true exactly when a step from s to s' is one of the action's steps. There are many notations (usually

called languages) for writing down these relations easily and clearly, but first-order logic underlies all of them. Example: `x:=y` is a way of writing the predicate $x' = y \wedge (\forall\, v$ **except** $x \mid v' = v)$; the value of `x` changes and all the other variables stay the same. There might be more than one possible next state if an action is nondeterministic, or none if it's blocked. The *behavior* of the system is just the set of possible sequences of steps.

A spec can be very partial, in which case it's often called a *property*; for example, it might just specify "no segfaults" by saying that any step that isn't a segfault is okay. As well as being partial, a spec can be *nondeterministic*: any of a set of results is acceptable; for example, a timing spec such as "Less than 200 ms". And often details *should* be left open. Eventual consistency just says that an update will appear in the state by the end of the next sync.

The code should *satisfy* (meet) the spec. That means that every visible behavior of the code is also a behavior of the spec. The "visible" is important; typically the code has internal state that's invisible, and often the spec does too.

Finding the right abstractions is the most important part of designing a system. A language gives you some built-in abstractions: strings, arrays, dictionaries, functions. These are good, but they are less important than the abstractions in the platform you are building on, such as files, networking, relational data, vectors and matrices, etc. And those in turn are less important than the abstractions that are specific to the application.

Which comes first, the spec or the code? In theory the spec should come first, since it reflects what you want done; this is called top-down design, and the code is a *refinement* of the spec. In practice they evolve together, because you can't tell what the spec should be until you see how it affects the code and the system's customers. The first ideas for a spec are usually much too closely tied to the code, and usually provide both more and less than the customers need.

2.1.1 *Safety and liveness*

Any spec is the conjunction of two parts:

- A *safety* spec, which says that nothing bad ever happens. If the code violates a safety spec the bad thing happens in a finite number of steps.
- A *liveness* spec, which says that something good eventually happens, usually that it's *fair*: every action allowed by safety eventually happens. No finite behavior can violate liveness, because there's always more time for the good thing to happen.

Usually safety is what's important, because "eventually" is not very useful; you care about getting a result within two seconds, and that's a safety property (violated after two seconds).

2.2 **Writing a spec**—KISS: Keep It Simple, Stupid.

Reality is that which, when you stop believing in it, doesn't go away. —Philip K. Dick[Q12]

How should you go about writing a spec? There are two steps:

(1) Write down the state of the spec (the **abstract** state).

You have to know the state to even get started, and finding the simplest and clearest abstract state is *always* worth the effort. It's hard, because you have to shake loose from the details of the code you have in mind and think about what your clients really need. The mental tools you need for this are the elementary discrete math of relations and a good understanding of the clients.

Often people say that the abstract state is not real; only the RAM bytes, the disk blocks and the machine instructions are real. I can't understand this; a physicist will say that only the quantum

mechanics of electrons in silicon is real. What they probably mean is that the spec doesn't actually describe the behavior of the system. This can happen in several ways:

- It can be wrong: the code does things the spec doesn't allow. This is a bug that should be fixed.
- It can omit important details: how accurate a sine routine is or what happens if there's a failure.
- It can omit unimportant details by being leaky. This is a matter of judgment.

For the file system example, the spec state has files F, directories D, nodes N and a D node $root$. The state is a map $s = N \to (F \textbf{ or } D)$ that gives the current contents of the nodes. A file is a pair $F = (sz\text{: Nat}, data\text{: }\textbf{array}\text{ Byte})$ and a directory is a (partial) function $D = Name \to N$. The D's must organize the nodes into a graph where the F's are leaf nodes and the D's form a tree rooted in $root$; an invariant on the state says this.

(2) Write down the spec actions: how each action depends on the state and changes the state.

Now you have everything the client needs to know. If you haven't done this much, you probably can't do a decent job of documenting for the client. Writing down the actions precisely is a lot more work, and you probably need a good notation (language) to do it clearly and concisely. The 2009 lecture notes for my MIT course 6.826, Principles of Computer Systems, have many realistic examples worked out in detail, unfortunately using a made-up language.[R32]

Good specs are hard. Each spec is a small programming language with its own types and built-in operations, and language design is hard. Also, the spec mustn't promise more than the code can deliver—not the best possible code, but the code you can actually write.

There is nothing special about concurrency, except that it makes the code (and perhaps the spec) nondeterministic: the current state doesn't determine the next step, because any thread that isn't blocked could provide it. Likewise there is nothing special about failures. A crash or the misbehavior of a component is just another action. Crashes cause trouble because they may destroy state that you would prefer to keep, and because they add concurrency that you don't have much control over. But these are facts of life that you have to deal with.

2.2.1 *Leaky specs and bad specs*

Specs are usually incomplete or *leaky*. Most notably, specs often don't say much about speed. Sometimes the spec needs to be leaky, in the sense that it exposes some internal secrets, to give clients the access they need to run fast. Being leaky is not necessarily a bad thing, and in general it's unavoidable. But there are other properties of a spec that *are* usually bad:

- *Complexity* is hard for the client to understand, and hard to code. It often comes from not following the state-and-actions recipe, or exposing facts about the code that should be secret.
- *Brittleness* makes the spec depend on details of the environment that are likely to change, or on details of how it is called that are easy to get wrong.
- *Errors* or *failures* in the code that the spec gives no way to report mean that the code won't satisfy the spec. A common example is a synchronous API that makes the code look local, fast and reliable even though it's really remote, slow and flaky.
- Similarly, *contention* or overload may keep the code from meeting the spec if there's no way to report these problems or set priorities.
- *De facto specs*, in either function or performance, happen when the code has properties that clients come to depend on even though they are not in the spec.

2.3 **Writing the code: Correctness—**Get it right

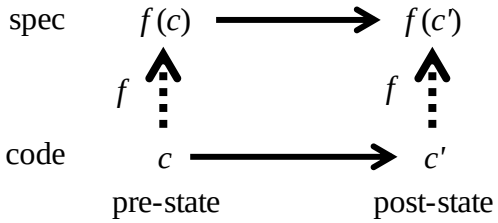
Smart software companies know that reliable software is not cost effective. ... It's much cheaper to release buggy software and fix the 5% to 10% of bugs ... people complain about. —Bruce Schneier[Q45]

Most of this paper is about how to write the code. But is the code correct? In other words, does it satisfy the spec? (You don't have a spec? Then the question makes no sense.) In theory this question has a yes-or-no answer. If

- the spec is a predicate that specifies every allowed action (step) of the system,
- the code precisely specifies every action that the system takes, and
- you know which parts of the state are visible to the client,

then correctness is a theorem: "Every visible code behavior is a spec behavior," either true or false.

If the theorem is true, a surprising fact is that it has a *simulation proof*: there is an *abstraction function* f from the code state to the spec state such that every code action $c \to c'$ from a reachable state has a matching spec action $f(c) \to f(c')$ with the same effect on the visible state (it's the identity if the action doesn't change any visible state).



This diagram is the inductive step in the proof that every visible code behavior is a spec behavior. You might need to add history or prophecy variables (or use an abstraction relation).[R1]

Following this script, once you have the spec (steps (1) and (2) above) and the code state and actions, there are two more steps to connect them:

(3) Find an abstraction function from code to spec state. Also find the invariants on the code state, that is, define the states that the code can reach; the proof only needs to deal with actions from reachable states. For example, if the code has a sorted array there will be an invariant that says so, and you need it to show that lookup actually works.

(4) Finally, do the actual simulation proof that every code action preserves the visible behavior and the invariants.

Step (4) is the only one that requires reasoning about *every* action in the code from *every* reachable code state, so it's by far the most work. Step (3) requires *understanding* why the code works, and it usually uncovers lots of bugs. Unfortunately, the only way to know that you've done it right is to do step (4), which is usually not worthwhile. But writing a spec is always worthwhile.

An alternative is *model checking*, which explores a small subset of the code's state space systematically, looking for behavior that violates the spec. This doesn't give any proof of correctness (unless there are so few behaviors that the checker can try all of them), but it finds a lot of bugs.[R38,R19]

2.3.1 *Types*

Types are a way to express some facts about your program that the machine can understand and check, in particular some stylized preconditions and postconditions. The idea is that

- a value v of type T has an extra `type` field whose value is T,
- an argument must have the expected type (the precondition): if a routine R expects a type T', $R(v)$ is an error unless $T = T'$ (or more generally, T is a subtype of T'),
- a routine's result has the expected type (the postcondition).

With dynamic types the type field is present at runtime; most often it's called a class. In a static system it's a "ghost" field not present at runtime, because every expression e has a type.

Why are static types good? For the same reason that static checking in general is good: the compiler can prove theorems about your program. Most of the theorems are not very interesting, since they just say that arguments have the right types. But the first draft of a program almost always has lots of errors, and most errors are pretty obvious. So type checking finds lots of bugs when it can't prove its trivial theorems.[R41]

2.3.2 *Languages*

What programming language should you use? There is no universal answer to this question, but here are some things to think about:

- How hard is it to write your program so that the language guarantees that it has a bullet-proof abstract state, in which a variable always has the expected type and only changes when it's explicitly written? Usually this means strong typing and garbage collection. JavaScript is bulletproof in this sense, C++ is not. A broken abstraction makes debugging much more difficult.
- Is the language well matched to your problem domain? Is it easy to say the things that you say frequently? Is it possible to say all the things that you need to say?
- How much static checking does the compiler do? A bug found statically is easier to handle.
- How hard is it to make your program efficient enough, or to measure how it uses resources?

2.4 **Modules and interfaces**—Keep it clean

The only known way to build a large system is to reinforce abstraction with divide and conquer: break the system down into independent abstractions called *modules*. The running code of a module is often called a *service*. The spec for each module does two things:

- it *simplifies* the client's life by hiding the complexity of the code (see above), and
- it *decouples* the client from the code, so that the two can evolve independently.

Thus many people can work on the system productively in parallel without needing to talk to each other. A really successful spec is like an hourglass: the spec is the narrow neck, with many clients above and many codes below, and it can live for decades. Examples: CPU ISAs (instruction set architectures such as x86 and ARM), file systems (Posix), reliable messages (TCP), names for Internet services (DNS), web pages (HTTP and HTML).

It's common to call the spec of a module its *interface*, and I'll do this too. Unfortunately, in common usage an interface is a very incomplete spec that a compiler or loader can process, giving just the data types and the names and (if you're lucky) the parameters of the operations, rather than what the actions do with the state. Even a good description of the state is often missing.

2.4.1 *Classes and objects*

A variation on modules attaches the spec and code to a data item, usually called an *object*. You choose a set of routines called *methods* that take the same type of data as their first argument, and package their specs into a single spec, here called a *classpec* (it's called an abstract base class in C++ and Python). The code for the classpec is a *class*, a dictionary that maps each method name to its code. An object of the right type that has the class attached is an *instance* of the class.

For example, the classpec `Ord T` might have methods `eq` and `lt`. If `x` is an instance of `Ord T`, then `x.eq(y)` calls the `eq` method in `x`'s class with arguments `(x,y)`. Adding methods to a class makes a *subclass*, which *inherits* the superclass methods; thus `Ord T` is a subclass of an `Eq T` class that has only the `eq` method. An instance of `Ord T` is also an instance of `Eq T`.

2.4.2 *Layers and platforms*

A typical system has lots of modules, and when a module's spec changes you need to know who depends on it. To make this easier, put related modules into a *layer*, a single unit that a team or vendor can ship and a client can understand. The layer only exposes chosen interfaces, and a lower layer is not allowed to call a routine in a higher layer. So a layer is a big module, normally a client of its *host*, a single layer below it, with one or more layers as its clients above it.

Usually you build a system on a *platform*, a big layer that serves a wider range of clients and comes from a different organization. Common platforms are a browser (the interface is a document object model accessed through JavaScript) or a database system (the interface is SQL), built on an operating system platform (Windows or Linux; the interface is kernel and library calls) built on a hardware platform (Intel x86 or ARM; the interface is the ISA). It's turtles all the way down: the hardware is built on gates and memory cells, which are built on transistors, which are built on electrons . Here is an example with all the turtles:

application	Gmail
web framework	Django
database browser	BigTable Chrome
operating system	Windows 10
virtual machine	VMware
ISA	X86
CPU hardware	AMD Ryzen 7 2700X
gates memory	TSMC 7 nm Micron MT40A16G4
transistors	7 nm finFET LPDDR4X-4266
electrons, quantum mechanics	

2.4.3 *Components*

Reusing pieces of code is like picking off sentences from other people's stories and trying to make a magazine article. —Bob Frankston[Q17]

It's harder to read code than to write it. —Joel Spolsky[Q47]

A module that is engineered to be reused in several systems is called a *component*. Obviously it's better to find a component that does what you need than to build it yourself (don't reinvent the wheel), but there are some pitfalls:

- You need to *understand* its spec, including its performance.
- You need to be confident that its code actually *satisfies* the spec and will be maintained.
- If it doesn't quite do everything that you want, you have to *fill in the gaps*.
- Your environment must satisfy the *assumptions* the component makes: how it allocates resources, how it handles exceptions, how it's configured, and the interfaces it depends on.

There are two ways to keep from falling into one of these pitfalls:

- Copy and paste the module's code into your system and make whatever changes you find necessary. This is usually the right thing to do for a small component, because it avoids the problems listed above. The drawback is that it's hard to keep up with bug fixes or improvements.

- Stick to the very large components usually called platforms. These have a viable business model (because it's impractical to write your own), there will only be a few of them to learn about, they encapsulate a lot of hard engineering work, and they stay around for a long time.[R30] A good library can also be a source of safe components that are smaller than a whole platform.

2.4.4 *Open systems*—Don't hide power. Leave it to the client.

The point of an abstraction is to hide how the code is doing its work, but it shouldn't prevent a client from using all the power of its host. An abstraction can preempt decisions that its clients could make; for example, its way of buffering I/O might keep a device from running at its full bandwidth. If it's an ordinary module, a client can always hack into it, but that's not an option if it's an operating system that isolates its clients, or if you want to keep taking bug fixes. The alternative is careful design that doesn't hide power, but gives clients access to all the underlying performance. *Scheduler activations* are an example; they are less convenient than threads, but give the client control over scheduling and context switching. *Exokernels* carry this idea further, moving most of the code of an OS platform into a *library OS* that the client can change if necessary.

Another way to expose an abstraction's power is to make it programmable, either by callbacks to client-supplied functions or by programs written in an application-specific instruction set. There are many examples of this:

- The SQL query language, a functional instruction set.
- Display lists and more elaborate programs for GPUs.
- Software-defined networking.
- Binary patching, first done in the Informer, a tool for instrumenting an OS kernel. It checked the proposed machine code patch for safety.[R18] Later there were binary modification tools[R46].

3 Goals and Techniques

3.1 Simple

Entities should not be multiplied beyond necessity. —William of Occam[Q37]
I'm sorry I wrote you such a long letter; I didn't have time to write a short one. —Blaise Pascal[Q39]
Everything should be made as simple as it can be, but not simpler. —Albert Einstein[Q16]
There are some insurmountable opportunities around. —Don Mitchell[Q34]
Simple things should be simple, complex things should be possible. —Alan Kay[Q25]

3.1.1 *Do one thing well*

Figure out how to solve one really tricky sticky problem and then leave the rest of the system straightforward and boring. I … call this the "rocket science" pattern. —Terry Crowley[Q10]

Design your system around a small number of *key* modules with simple specs and predictably good performance. If you're lucky you can get these modules from your platform or from a library. If not, you have to build them yourself, but your goal should be the same. Finding this system design and building the key modules is hard work, but it's rewarded throughout the system's life because you can concentrate on the customers' needs; the rest of the code is easy to change, since it won't need any real cleverness. A successful key module will grow over time, improving performance with better algorithms and adding a few features, but building on a solid foundation. Make it fast, rather than general or powerful; then the client can program the function it wants.

A wide range of examples illustrate this idea:

- The inode structure in a file system represents variable-length byte strings, even very large ones. Many variations fit in: variable-length extents (ranges of disk blocks) to keep the index small, sharing parts of the byte string for copy-on-write, logs for crash recovery.
- The Unix version 6 operating system is an amazing example, notably separating file directories from inodes, and connecting applications by byte streams through the shell.
- The basic Internet protocols (TCP and UDP) provide reliable and best-efforts communication among billions of nodes.
- The simplicity of the BitBlt interface made it the standard for raster display applications.
- The Domain Name System is an eventually consistent hierarchical name space that's the foundation of Internet naming, for the web, email, and many other things. It maps a path name such as `csail.mit.edu` into a set of small "records".
- Relational databases structure very large amounts of data as tables with named columns.

Often a module that succeeds in doing one thing well becomes more elaborate and does several things. This is okay, as long as it continues to do its original job well.

3.1.2 *Brute force*

Computers are fast, and specialized hardware is even faster—take advantage of it. Exhaustive search (perhaps only up to some "depth") is a simple brute force technique. It's $O(n)$, and often n is not too big, so always consider it first. Examples: `grep` over a file, model checking, many optimization problems, and a host of attacks on security measures such as password guessing. It's also the only way to query a database if you don't have an index. It works best when you have locality.

Broadcast is a second example of brute force. It is to routing as exhaustive search is to indexing, and it scales badly. In networking, though, you often need a broadcast to get started. A third example is polling to find pending work, in contrast to notification.

3.2 Timely

Building a timely system (one that ships soon enough to meet your time-to-market needs) means making painful choices to give up features and dependability. If it's extensible you can add features later; adding dependability is harder. It's easier to make approximate software timely.

»The web. Perhaps the biggest reason the web is successful is that it doesn't have to work. The model is that the user will try it again, switch to an alternative service, or come back tomorrow. It's quite rare to find a web service that is precise. For example, there's no spec for a search engine, since you can't write code for "deliver links to the 10 web pages that best match the customer's intent", and indeed engines are ruthless about ignoring parts of the Internet in order to deliver results faster.

»Agile software. A more surprising example comes from a major retail web site, where the software is developed as hundreds of modules. Each module is developed by a small team that has complete control over the specs and code. Any module can call any other module. There is no integration testing or release control. Not surprisingly, it's common that a module fails to deliver expected or timely results; this means that its caller must be programmed defensively. Retail customers may notice that some of the web pages they see are incomplete or wrong—the only page that really must be correct is the one with the "Place Your Order" button. Of course, credit card processing uses precise software.

3.3 Efficient

An efficient program is an exercise in logical brinksmanship. —Edsger Dijkstra[Q15]

The greatest performance improvement of all is when a system goes from not-working to working —John Ousterhout[Q38]

Efficiency is about doing things fast and cheaply. Most of what I have to say about it is in the ABCs below: **A**lgorithms, **A**pproximate, **B**atch, **C**ache, **C**oncurrent, **C**ommute, **S**hard/**S**tream. But first some generalities.

3.3.1 *Before the ABCs*

It's tricky to write an efficient program, so don't do it unless you really need the performance. If a shell script is fast enough to solve your problem, by all means use a shell script.[R7] If you do optimize, remember the rule: make the code correct first and then make it fast.

The resources you are trying to use efficiently are computing, storage, and communication. The dimensions are time and space: how long something takes and how many resources. For time the parameters are *bandwidth* (or throughput) and *latency* (or response time). Latency is the time to do the work (including communication) plus the time spent waiting for resources because of contention (queuing). To evaluate a design idea, start by working out roughly how much latency, bandwidth and memory capacity it consumes to deliver the performance you need. Then ask whether with optimistic assumptions, that much could be available. If not, that idea is no good; if so, the next step is a more detailed analysis of the possible bottlenecks.

If you can divide the work into independent parts, you can use concurrency to trade more resources (more bandwidth) for less latency. With enough independence the only limit to this is the budget and the number of parts, as cloud services for search, email, etc. demonstrate. Likewise, a cache lets you trade locality and bandwidth for latency: if you use a fraction f of the data, you need $1/f$ extra bandwidth.

Fast path and bottlenecks

There are two basic ways to reduce latency: concurrency and *fast path*—do the common case fast, leaving the rare cases to be slow. For caching, the fast path is a cache hit. *Amdahl's Law* governs the performance of fast path: if the slow path has probability $p \ll 1$, the fast path takes time f, and the slow path takes time $s \gg f$, then the average time is $f + ps$. The *slowdown* from the slow path is $(f + ps)/f = 1 + p(s/f)$. Thus a RAM cache with $p = 1\%$ (99% hits) and $s/f = 100$ (1 ns to cache, 100 ns to RAM) is $2\times$ slower than a hit every time.

Amdahl invented his law to describe the limit on speedup from concurrency. Here the slow path is the part that must be done serially. The *speedup* from the concurrent fast path is $s/(f + ps) = 1/(f/s + p)$. With n-way concurrency $f = s/n$ and the speedup is $1/(1/n + p)$ For large n this is just $1/p$. If $p = 1\%$ (only 1% is serial), the maximum speedup is $100\times$, no matter how much concurrency there is. Whether you want to think of the result as a speedup or slowdown depends on your expectations.

Almost the opposite of a fast path is a *bottleneck*, the part of the system that consumes the most time. Look for the bottleneck first. Usually you don't need to look any farther; it dominates the performance, and optimizing anything else wastes time and adds complexity. Once you've found it, find a fast path that alleviates it. In other words, design your code to use it as little as possible, and measure and control how it's used.

Predictable performance

That, Sir, is the good of counting. It brings everything to a certainty, which before floated in the mind indefinitely. —Samuel Johnson[Q22]

What you measure is what you'll get. —Dan Ariely[Q2]

Your guess about where the time is going is probably wrong. Measure before you optimize. If you depend on something unpredictable, measure it in the running system and either adapt to it, or at least report unexpected values so that developers or operations staff can tell what's going on.

It's often not enough for a spec to describe the state that the program can name. Resources must be part of the state, including real time, and an action must say what resources it consumes,

especially how long it takes. Ideally this won't depend on the environment or on parameters of the action, but often it does and you need to know how in order to use the action effectively. A module can control many aspects of its performance: internal data structures and algorithms, optimization, compression, etc. But the environment controls other aspects: latency and bandwidth to storage, between address spaces and between machines. This can change as the clients' demands or the underlying platform change, and a robust application must either adapt or report that it can't.[R16]

Don't try to be precise; it's enough to know how to avoid disaster, as in paging, where you just need to keep the working set small enough.

Network access in general is very unpredictable (except in a data center, where the environment is usually tightly managed) and you can't control it very well, so it's best to work only on local data (which might be stale) when responding to a user input, unless it's very obvious to the user that the network is involved, for example in a web search. This means that the UI should communicate asynchronously with anything that might be slow, using some form of eventual consistency to process the response when it finally comes.

Locality—Keep it close

Because communication is expensive and memory hierarchies are deep, keep the data close to the computation. The L1 cache is the closest it can get, but in general you just want it close enough that moving it to the computation is cheap enough. The most important two strategies are:

- Keep the parts that run concurrently as independent as possible, to minimize communication.
- Make the data smaller, so that more of it is local and there's *less* to communicate. Try to get by with a summary of the full dataset.

Contention

If there aren't enough resources to process the instantaneous load there will be *contention*, which shows up as *queuing* for access to a resource and increases the latency. It's hard to understand queuing in general, but the simplest case is easy and important: if a resource is busy (utilized) for u seconds per second and tasks arrive randomly, then a task that uses it for a second will take $1 / (1 - u)$ seconds. For example, at $u = 90\%$ it takes 10 seconds—ouch!

3.3.2 *Algorithms*

[In many areas] performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed. —PCAST[Q40]

Fancy algorithms are slow when N is small, and N is usually small. —Rob Pike[Q41]

There's been a lot of work both on devising algorithms for important problems and on analyzing their performance. The analysis bounds the running time $t(n)$ asymptotically as the problem size n grows: $t(n) = O(n)$ means that there's a constant k such that $t(n) \leq kn$ as $n \to \infty$. Anything worse than $O(n \log n)$ is bad unless n is sure to be small, but this is not the whole story.

- There can be a large fixed overhead (which is bad when n is small), and k can also be large.
- You might care about the average rather than the worst case.

It's usually best to stick to simple algorithms: a hash table for looking up a key, a B-tree for finding all the keys in a range, a DHT for strong fault tolerance. Books on algorithms tell you a lot more than you need to know. If you have to solve a harder problem from a well-studied domain such as numerical analysis or graph theory, look for a widely-used library. If n is really large (say the Facebook friends graph), look for a randomized *sublinear* algorithm with time $< O(n)$; for example, the median of a large set of size n is close to the median of a random subset of size $\log n$.

3.3.3 *Approximate*—Flaky, springy parts

It is better to have an approximate answer to the right question than an exact answer to the wrong one. —John Tukey[Q49]

Very often you don't need an exact answer; a good enough approximation is fine. This might be "within 5% of the true answer" or "the chance that the answer is wrong is less than 1%." If the "chance" in the latter is truly random, you can make it .01% by doing it twice. Sometimes the answer is just a guess, which you need to validate by watching the running system.

You can approximate the *analysis* rather than the solution; this is called "back of the envelope" analysis, and usually it's all you need. How to do it: find the few bottleneck operations that account for most of the cost, estimate the cost and the number of times you do each one, multiply and add. For example, for a program that does 10^{10} memory operations, has a cache hit rate of 95%, and runs on a machine with RAM access time of 100 ns, the cache is the bottleneck and it will take about $10^{10} \times .05 \times 100/10^9 = 50$ sec.

It often pays to *compress* data so that it's cheaper to store or transmit. The most powerful compression produces a *summary* that is much smaller than the input data.

- A *sketch* captures the most important things about the input. Examples: a low resolution version of an image; a vector of hashes such that two similar documents have nearby vectors[R11].
- A *Bloom filter* is a bit vector that summarizes a set of inputs for testing membership. If a new input is in the set the filter will say so; if it's not, the filter will wrongly say that it is with some probability f. With 10 filter bits per set element $f < .01$, with 20 filter bits $f < 10^{-4}$.[R36]
- *Sampling* a data set summarizes it with a much smaller set whose properties are good approximations to properties of the original. Often $\log n$ samples from a set of size n is enough.
- A *classifier* tells you some property of the input, for example, whether it's a picture of a kitten.
- A *Merkle tree* summarizes the subtree rooted in a node by a hash of the node's children. If the tree is balanced, it takes only $\log n$ operations to check that a node is in the root's hash.
- *Abstract interpretation* summarizes the dynamic behavior of a program by making it static, replacing each variable with a simpler abstract one whose value is a constant.

Approximate behavior

Another kind of approximation works on a program's *behavior* rather than its data.

- A hint is a value that might be what you want, but you need to check that it's valid.
- *Exponential backoff* is a distributed algorithm in which each node responds to an overload signal by decreasing its offered load exponentially. Examples: ethernet, Internet TCP, Wi-Fi.
- A randomized algorithm gives an answer with probability p of being wrong. If p isn't small enough, repeat n times and the chance of being wrong is p^n, as small as you like.
- Eventual consistency lets applications operate on stale data.
- Agile software development approximates the system spec to get something running quickly for users to try out. Their reactions guide the evolution of the spec.

Hints

A hint (in the technical sense) is information that bypasses an expensive computation if it's correct; it's cheap to *check* that it's correct, and there's a *backup* path that will work if it's wrong. There are many examples of hints scattered through the paper, but here are some general patterns:

- An approximate index points to an item in a large data set that contains a search term, or more generally that satisfies a query. To check the hint, check that the item does satisfy the query.

- A *predictor* uses past history to guess something. A CPU predicts whether a conditional branch will be taken; the check is to wait for the condition, the backup is to undo any state changes.[R20]
- Routing hints tell you how to forward a packet or message. The backup is rerouting.

3.3.4 *Batch*—Take big gulps

Whenever the overhead for processing b items is much less than b times the overhead for a single item, batching items together will improve performance. If the batch cost is s, the cost per batched item is f and the batch size is b, the total cost is $s + fb$ and the cost per item is $f + s/b$. This is just the fast path formula $f + ps$, with $b \approx 1/p$; bigger batches are like a smaller chance of taking the slow path. Batching increases bandwidth, but it also increases latency.

The opposite of batching is *fragmenting*, artificially breaking up a big chunk of work into smaller pieces. This is good for load-balancing, especially when either the load or the service time is bursty. Fragmenting bounds the increase in latency, and it also keeps small jobs from getting stuck behind big ones. Fragments in a network are called packets.

Here are some examples of batching:

- A cache with a line size bigger than the size of the data requested by a load instruction.
- Minibatches for deep learning; each minibatch trains a set of weights that fits in the cache.
- Group commit, packing the commit records for many transactions into one log record.
- Indexing, which pays a big cost upfront to build the index so that later queries will be fast.
- Epochs, batching deletions or other changes to reduce syncing, as in read-copy-update[R35].

3.3.5 *Cache*

The idea of caching is to save the result of a function evaluation $f(x)$. The result is an overlay of the partial function defined by the cache on the base function f. The best-known application is when $f(x)$ is "the contents of RAM location x"; CPUs implement this in hardware. File and database systems do the same in software, keeping disk pages in RAM. A cache for a storage system is lazy partial replication, done for performance rather than fault tolerance.

As important are the software indexes of databases and search engines, where $f(x)$ is "the table rows or documents matching x". Without an index you have to scan the entire database to evaluate these functions.

If f is not pure ($f(x)$ depends on the state as well as on x), then when state changes cause $f(x)$ to change you must either tolerate stale cache values, or invalidate or update a cache entry. This requires that the source of the change either

- sends a *notification* to any cache entries that depend on it, or
- broadcasts every state change, and the cache watches the broadcasts.

For a RAM cache a change is a store to an address in the cache, and the two techniques are called *directory* and *snooping*.

Here are some other examples of caching a function:

- Network routing tables that use the destination address to tell you where to send a packet. These are updated lazily by a routing protocol such as BGP, OSPF, or ethernet switching.
- Shadow page tables in virtual machines, which cache values of the mapping $(guest\ VM, virtual\ address) \rightarrow host\ physical\ address$, the composition of $guest\ VA \rightarrow guest\ PA$ and $guest\ PA \rightarrow host\ PA$.
- Materialized views in a database, which cache the table that's the result of a query.

3.3.6 *Concurrency*—S^3: shard, stream or struggle. Make it atomic.

Now that single-stream general-purpose processors are not getting faster[R34], there are only three ways to speed up a computation: better algorithms, specialized hardware and concurrency. Only the latter is reasonably general-purpose, but it has two major problems:

- It's hard to reason about concurrent computations that make arbitrary state changes, because the concurrent steps can be interleaved in so many ways. Hence the S^3 slogan.
- To run fast, data must be either immutable or local, because when a remote variable changes, getting its current value is costly. Fast computations need P&L: parallelism and locality.

The other reason for concurrency is that part of the computation is slow. Disk accesses, network services, external physical devices, and user interaction take billions of processor cycles. When the slow part is done it has to get the attention of the fast part, usually by some form of *notification*: interrupt a running thread, wake up a waiting thread, post to a queue that some thread will eventually look at, or run a dispatcher thread that creates a new thread.

Sharding is really easy concurrency that breaks the state into n pieces that change *independently*. A single thread touches only one shard, so the steps of threads that touch different shards don't depend on the interleaving. A *key* determines which shard to use. The simplest example is disk striping: a few bits of the address are the key that chooses the disk to store a given block, and all the disks read or write in parallel. Fancier is a sharded key-value store with ordered keys; $n-1$ *pivot* values divide the keys into n roughly equal chunks. To look up a key, use the pivot table to find its shard.

Often there's a *combining function* for results from several shards. A simple example is sampling, which just takes the union of a small subset from each shard

Streaming is the other really easy kind of concurrency: divide the work for a single item into k sequential steps, put one step on each processor, and pass work items along the chain. This scheme generalizes to *dataflow*, where the work flows through a DAG. The number of distinct processing steps limits concurrency. Use batching to amortize the per-item overhead.

Map-reduce combines these two techniques, alternating a sharded map phase with a combining reduce phase that also redistributes the data into shards that are good for the next phase. It can reuse the same machines for each phase, or stream the data through a DAG of machines.

Beyond shards and streams—struggle

Do I contradict myself? Very well then I contradict myself. (I am large, I contain multitudes.) —Walt Whitman[Q54]

I may be inconsistent. But not all the time. —Anonymous

If you can't shard or stream, you will have to struggle. It helps to think in terms of showing that a general nondeterministic program is correct, and then letting performance constrain the choices: scheduling (including timeouts, interleaving, losses), table sizes, etc. If the abstract state that your language provides is not bulletproof (type and memory safe) you'll struggle more.

There are five kinds of concurrency; the first two provide *consistency*, the same result as running the actions in some sequential order.

- **Really easy**: pure sharding or streaming. Either actions are *independent*, sharing no state except when they are combined, or they communicate only by *producer-consumer* buffers.
- **Easy**: make a complex action *atomic* so that it behaves as though the entire action happened sequentially (serially). To do this, group the actions into sets that break atomicity if they run concurrently, such as reads and writes of the same variable. Have a *lock* variable to protect each set, with the rules that:

  - Before running a protected action, a thread must acquire its lock.
  - Two locks in different threads *conflict* if the actions they protect don't commute (for example, writes of the same variable don't commute with reads or other writes).
  - A thread must wait to acquire a lock if another thread holds a conflicting lock.
- (**Nuisance**: discussed in the full paper.)
- **Hard**: anything else. With hard concurrency you can do a formal **proof** or have a **bug**.
- **Eventual**: all updates commute, so you get the same eventual result regardless of the order they are applied, but you have to tolerate stale data. This is easy to code:
  - Make updates commute. If they are blind writes, time-stamp them; last writer wins.
  - Broadcast the updates to all the nodes.

  It's also highly available, since you can always run using only local data. The apps pay the piper: they must deal with stale data. There are many examples: name services like DNS (which has no sync), key-value stores like Dynamo, and "relaxed consistency" multiprocessor memory.[R4]

An important special case of easy concurrency is *epochs*, a batching technique that maintains some invariant on the state except at the step from one epoch to another. An epoch is a special case of locking that holds a global lock on certain changes throughout the epoch, so that the changes can only occur when the epoch ends and releases the lock. The code follows these rules by convention; there's no lock variable that's acquired and released. Most often the change that is locked is deleting an object, so that code that gains access to an object during the epoch knows that the object won't disappear unexpectedly. For this to work well it has to be okay to defer the deletions. Sometimes the global lock prevents *any* changes to certain objects, keeping them immutable during the epoch.

Locks don't work well in a distributed system because they don't play nice with partial failures. Leases can be a workaround. The only meaningful content in an asynchronous message is facts that are *stable*: once they are true, they are true forever.

A good rule of thumb is the *scalable commutativity rule*: if the specs of two actions commute, then you can write code in which they run concurrently, which is important for keeping all the cores busy on modern CPUs. For example, Posix file `open` returns the *smallest* unused file descriptor; if it returned an *arbitrary* unused descriptor, two `open`s could commute.[R13]

3.4 Adaptable

There are many things that your system might need to adapt to:

- Changes in the **clients' needs**: new features or data formats, higher bandwidth, lower latency, better availability.
- Changes in the host **platform**: new interfaces or versions, better or worse performance.
- Changes in **regulation** or in security **threats**: privacy or other compliance requirements, data sovereignty, broken cryptography, new malware.
- Changes in **scale**, from 100 clients to 100 million or from storing text to storing video.

Such changes may force major rework, but usually a well-designed system can adapt less painfully.

The keys to adapting to functional changes are modularity and extension points in the design. The keys to adapting to scaling are modularity, concurrency, and automation.

Changes in interfaces cause a compatibility problem: unless the client and the service spec change at the same time, there's a mismatch. One solution is to make the new spec a superset of the old one. This has worked well for ethernet, the Internet, many ISAs, some programming

languages, and basic HTML; 40-year-old clients still work. The other solution is a form of indirection: an *adapter* or *shim* that satisfies the old spec and is a client of the new one. When the new one is dramatically different this is called *virtualization*.

3.4.1 *Scaling*

Expanding on the catchwords above, scaling requires:

- Modularity for algorithms, so it's easy to change to one that scales better.
- Concurrency that scales with the load by sharding: the work for different clients is independent and all communication is asynchronous.
- Automating everything, so that a human never touches just one machine (except to replace it if the hardware fails). This means fully automating both fault tolerance and operations.

The independent shards sometimes have to come back together. There are two aspects to this:

- Combining the independent outputs or synchronizing the shard states.
- Naming the shards, using big random numbers (which must be indexed) or path names.

If the shards already exist, use federation to put them into a single name space by making a new root with all of them as children.

- In a file system this is called *mounting*, and they stay independent.
- In a source code control system the shards are *branches* and synchronization is *merging*.

3.4.2 *Inflection points*—Seize the moment. Ride the curve.

History never repeats itself but it rhymes. —John Robert Colombo[Q8]

Why do great new technologies often fail? They are great when compared with the current incarnation of the boring old technology, but during the 3 to 5 years that it takes to ship the new thing, the old thing improves enough that it's no longer worthwhile to switch. This typically happens with new hardware storage technologies, such as thin film memories and optical disks.

The reverse happens when a new idea has some fundamental advantage that couldn't be fully exploited in yesterday's world, but conditions have changed so that it now pays off.

- Packets replaced circuits for communication when the computing needed to do the switching got cheap enough, and bandwidth got cheap enough for bursty data traffic to overwhelm voice.
- Ted Nelson invented the web in the 1960s (he called it hypertext), but it didn't catch on until the 1990s, when the Internet got big enough to make it worthwhile to build web pages.

3.5 Dependable

The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich find most hard to pay. —Tony Hoare[Q21]

A system is dependable if it is:

- **Reliable**—it gives the right answers in spite of partial failures.
- **Available**—it delivers answers promptly in spite of partial failures.
- **Secure**—it's reliable and available in spite of malicious adversaries.

The secret of reliability and availability is fault tolerance by *redundancy*: doing things independently enough times that at least one succeeds. Redundancy can be in time or in space.

- Redundancy in **time** is *retry* or *redo*: doing the same thing again. You have to detect the need for a retry, deal with any partial state changes, make sure the inputs still available, and avoid confusion if more than one try succeeds. The main design tool is end-to-end validation.

- Redundancy in **space** is *replication*: doing the same thing in several places. The challenges are giving all the places the same input and making the computation deterministic so that the outputs agree. The main tool is *consensus*.

It's very important for the redundancy to mostly use the *same* code as the normal case, since that code is tested and exercised much more, and hence has many fewer bugs. And of course redundancy won't do any good if a deterministic bug (a *Bohrbug*) caused the failure. On the other hand, many bugs are infrequent nondeterministic *Heisenbugs*, usually caused by concurrency.[R24]

Redundancy by itself is not enough; you also need *repair*. If one of two redundant copies fails the system continues to run, but it is no longer fault-tolerant. Similarly, if a component is failing half the time and a retry triples the cost, the operation takes six times as long as it should.

The idea of redundancy is to have no *single points of failure*. No single point of failure means a distributed system, which inherently is concurrent and has *partial failures*. This means that there are a lot more unusual states, which is why a distributed system is harder to get right than a centralized one, in which many errors just reset the whole system to a known state . A Bohrbug is also a single point of failure, unless the redundancy includes different code.

»Arpanet partitioning. On December 12, 1986, New England was cut off from the Arpanet for half a day. The map showed that there were seven connections to the rest of the network, but not that all seven of them went through the same fiber-optic cable between Newark and White Plains.[R25] In theory carriers can now guarantee that two connections share no physical structure.

»Cellphone disconnected. I tried to call a friend at the Microsoft campus on his office phone. It didn't work because it was a VOIP phone and his building's Internet connection was down. So I tried his cellphone, and that didn't work either because his building had a local cell tower, which used the building's Internet to connect to the wireless carrier and was too stupid to shut itself off when it could no longer connect.

3.5.1 *Correctness*

The best way to get your code to be correct is to keep it simple, and the best way to do that is to structure your system so that the most critical parts of the spec depend only on a small, well-isolated part of the code. This is the *trusted computing base* (TCB), invented to keep computer systems secure but applicable much more broadly. It's a good idea, but there are some difficulties:

- Keeping the TCB isolated from bad behavior in the rest of the system.
- Keeping the "most critical" parts of the spec from growing to include all of it.
- Maintaining the structure as spec and code change.

The single best tool for making a TCB is the *end-to-end* principle[R43]; its underlying idea is that the client is in control. More specifically, if you can easily *check* whether an answer is correct and you have a *backup* procedure, then the code that *generates* the answer doesn't need to be part of the TCB, and indeed doesn't need to be reliable. To use it you need a check for failure; if you're just sending a message this is a strong checksum of the contents, and a timeout in case it never arrives. The checksum also works for storage.

You probably don't want to give up if the check fails, so you need the backup; end-to-end says that this decision is up to the client, not the abstraction. You need to undo any visible state change caused by the failure. After that, if the failure is nondeterministic retrying is a good backup. The canonical example is TCP, which makes the flaky best-efforts packet service of the raw Internet into a reliable congestion-controlled byte stream. Other possibilities are trying something more expensive, especially if it was a hint that failed, or running in a degraded mode such as eventual consistency (with or without notice to the client). There may be no backup; encryption, for example, can't prevent a denial of service attack, though it can guarantee secrecy and integrity.

3.5.2 *Retry*—Do it again

If you can tell whether something worked, and there's a good chance after it fails that it will work better the second time, then retry is the redundancy you want. This applies especially to networking, where often you don't have good control of the communication, and even if you do it's much cheaper to tolerate some errors. Retry is based on the end-to-end principle, and in most applications you expect it to succeed eventually unless the network is partitioned or the party you are talking to has failed. Retry is a form of fast path: success on the first try is the fast path, with cost f, and the cost of the slow path is $s = r(1 + p + p^2 + \cdots) = r/(1 - p)$, where r is the time for one retry (the time it takes to detect a failure, usually a timeout, and try again) and p is the chance of failure. The slowdown caused by retries is $1 + p(s/f)$. For example, if a retry costs $10 \times$ a success ($r = 10f$), then you need $p \ll 10\%$ to make the cost of retry small.

If p is too big (perhaps because the chance of corrupting a message bit is too big), forward error correction (an error-correcting code) can make it smaller. An alternative is to reduce the number of bits by breaking the work into smaller chunks that fail and retry independently.

A retry that succeeds is supposed to yield the same final state as a single try; this is *idempotence*. Some actions are intrinsically idempotent, notably a *blind write* of the form $x \coloneqq \text{constant}$. To make an arbitrary action such as $x \coloneqq x + 1$ idempotent, make it *testable*: give it a unique ID, remember the IDs of completed actions (often the versions of variables), and discard any redundant retries. In communication this is *at-most-once* messaging ("at most" rather than "exactly" because the message is lost if the sender fails or gives up). The reason that the payment pages of online commerce often say "don't hit the back button and retry" is that they are doing this wrong.

A different form of retry is redo recovery from a log after a crash. If every pair of actions a and b in the log either commute ($a; b = b; a$) or absorb ($a; b = b$), then redoing prefixes of the log repeatedly (which happens if there are crashes during recovery), followed by redoing the whole log, is equivalent to redoing the whole log once. This is *log idempotence*. A blind write absorbs an earlier write to x and commutes with a write to any other variable. A testable action absorbs itself.

3.5.3 *Replication*—Do it twice

A *replicated state machine* (RSM) is a way of doing a fully general fault-tolerant computation using the ideas of being and becoming. You make several replicas of the host running the same code, start them in the same state, and feed them the same sequence of deterministic commands. Then they will produce the same outputs and end up in the same state. Any of the outputs will do as the output of the RSM, or you can vote if you have at least three replicas and want to protect against the possibility that a minority is Byzantine.

Of course there are some complications:

- The replicas must all see the same sequence: they must all agree about the first command, the second command, etc. The *Paxos* algorithm for distributed asynchronous consensus does this; it guarantees that replicas will never disagree about commands, and it makes progress as long as a suitable quorum of replicas can communicate for long enough.
- The commands must be deterministic; this requires some care.
- If a replica fails, you can redo the whole sequence of commands from scratch, or copy the state of some other replica and redo recent commands.

Reads must go through the RSM as well, which is expensive. To avoid this cost, use the fact that physics provides a reliable communication channel called real time. One replica takes out a time-limited lock called a *lease* on part of the state through the RSM; this stops anyone else from

changing that state. Drawbacks are that the leaseholder can be a bottleneck, and if it fails everyone must wait for the lease to expire.

The usual way to do replication is as *primary-backup*: one replica is the primary, chosen by the RSM, and it has a lease on the whole state so that it can do fast reads and batch many writes into one RSM command. The backups see all the writes because of the RSM, and they update their state to be ready in case the primary fails. The RSM needs three replicas, but they only need to store the commands, not the entire state.

Replication can improve performance as well as fault tolerance, since you can read from any replica that you know is up to date. This only helps if there are a lot more reads than writes, since replicated writes are more costly.

»Ariane 5. The first flight of the European Space Agency's Ariane 5 rocket self-destructed because both inertial reference system computers failed. The computers shut down because of an uncaught exception caused by an overflow. Shutdown seemed reasonable to engineers familiar with random hardware failures rather than software Bohrbugs.[R8]

3.5.4 *Detecting failures: real time*

Real time is not just for leases. It's the only way to detect that a service is not just slow but has failed—it hasn't responded for too long. Another way is for the service to tell you about it, but it might be wrong or dead. How to decide how long is too long? Choose a timeout, and when it expires either retry or report the problem. For a client device the report goes to the human user, who can decide to keep trying or give up. For a service it ultimately goes to the operations staff.

How do you choose a timeout? If it's too short there will be a lot of unnecessary retries, failovers or whatever. If it's too long the overall system latency will be too long. If the service reports the progress it's making, that might help you to choose well.

This story applies to a *fail-stop* system, which either satisfies its spec or does nothing. After a *Byzantine* failure the system might do anything. These are trickier to handle, and out of scope here.

3.5.5 *Recovery and repair*

It's common to describe availability by counting nines: 6 nines is 99.9999% available, which is half a minute of downtime per year. A good approximation is $MTTR/MTTF$, mean time to repair over mean time to failure (how long the system runs before it fails to serve its clients promptly enough). When part of a fault-tolerant system fails, $MTTR$ is the time to fail over to a redundant component, not the time to fix the failing part. In a well-engineered system failover is less than the specified response time, so the *system* doesn't fail at all; this is why it's important to make failover fast. Repair is also important.

»Memory errors. At Xerox Parc in 1971 we built a medium-sized computer called Maxc, using the new Intel 1103 1024-bit dynamic RAM chip. We didn't really know whether this chip worked, but with single bit error correction we never saw any failures in the running system. So we used the same chips in the Alto, but we decided to just have parity. Everything was fine until we ran the first serious application, the Bravo full-screen editor, and we started to get parity errors. Why? It turned out that 1103's are pattern-sensitive. Although Maxc hardware reported a corrected error, there was no software to read the reports, and there were quite a few of them. Lesson: Do repairs.

We got the problem under control using a random memory test program. Two years later we built the Alto 2, using 4k RAM chips and error correction. The machine seemed to work flawlessly, but after another two years we found that in one quarter of the memory neither error correction nor parity worked at all, because of a design error. Why did it take us two years to notice? The 4k chips were much better than 1103's, and most bits in RAM don't matter much. This is why consumer PCs don't have parity: chips are pretty reliable, and parity errors hurt the PC manufacturer, but if random things happen Microsoft gets blamed. Lesson: Different parties may have different interests.

3.5.6 *Transactions*—Make it atomic

If a complex action is atomic (either happens or doesn't), it's much easier to reason about. The slogan for this is ACID: Atomic, Consistent, Isolated, Durable.

- Atomic: Redo recovery makes it atomic with respect to *crashes*: after a crash either the whole action has happened, or none of it.
- Consistent: The transaction can decide to *abort* before committing, which undoes any state changes and so makes it atomic with respect to its *own* work. So it can make changes fearlessly, only needing to leave the system in a good state (consistent) when it commits..
- Isolated: The locks of easy concurrency make it atomic with respect to *concurrent* actions.
- Durable: Changes made by a committed transaction are written to persistent storage, usually in several copies, so that they survive anything short of a truly catastrophic failure.

Transaction processing systems ensure all these properties by draconian control over the transaction's application code.

»Pixie dust. Transactions are the pixie dust of computing. They take an application that understands nothing about fault tolerance, concurrency, undo, storage or load-balancing, and magically make it atomic, abortable, immune to crashes, and easy to distribute across a cluster of machines.

3.5.7 *Security*

But who will watch the watchers? She'll begin with them and buy their silence. —Juvenal[Q23]
If you want security, you must be prepared for inconvenience. —Gen. Benjamin Chidlaw[Q7]

Computer security is hard because of the conflict between *isolation* and *sharing*. People don't want outsiders to mess with their computing, but they do want to share data, programs and resources. In the early days isolation was physical and there was no sharing except by reading paper tape, punch cards or magtape. Today there's a lot more valuable stuff in your computers, and the Internet enables sharing with people all over the world. The job of security is to say "No," and people like to hear "Yes," so naturally they weaken the security until they actually get into trouble.

Here are the most important things to do for security (which all add inconvenience):

- Focus: figure out what you really need to protect.
- Lower aspirations: secure only things so important that you'll tolerate the inconvenience.
- Isolation: sanitize outside stuff to keep it from hurting you, or don't share dangerous stuff.
- Whitelisting: decide what you do trust, rather than blacklisting what you don't.

It's traditional to describe the goals of security as *confidentiality*, *integrity* and *availability*; the acronym is CIA. The mechanisms of security are *isolation* and the gold standard of *authentication* (who is making a request), *authorization* (who is allowed access to a resource), and *auditing* (what happened). A decentralized system has the additional problem of establishing *trust*, which you do by indirection: you come to trust someone by asking someone else that you already trust. Thus to answer questions like, "What is the public key for `billg@microsoft.com`," you trust a statement from `microsoft.com` that says, "The public key for `billg@microsoft.com` is *K*, valid through 3/15/2019."[R31]

What are the points of failure? For security they are called a *threat model*, especially important because there are so many possible attacks (hardware, operating system, browser, insiders, phishing, …) and because security is fractal: there's always a more subtle attack. For example, how do you know that your adversary hasn't hacked the BIOS on your PC, or installed a Trojan Horse in the hardware?[R53] So you need to be very clear about what you are defending against and what you are not worrying about. The TCB is the dual of the threat model; it's just what you need to defend

against the threats. The end-to-end principle makes the TCB smaller: encryption can make a secure channel between the two ends, so that the stuff in the middle is not a threat to secrecy or integrity.

Code for security is often tricky, so don't roll your own. For secure channels, use TLS. For parsing text that is going to be input to complex modules like SQL or the shell, use standard libraries to defend against SQL injection and similar attacks. Similarly for encrypting data; it's easy to make mistakes in coding crypto algorithms, managing keys and blocking side channels.

3.6 Yummy

The Mac is the first personal computer good enough to be criticized. —Alan Kay[Q26]

A system is much easier to sell if it's yummy, that is, if customers are enthusiastic about it. There are some good examples:

- Apple makes consumer products that people love to use, sacrificing functionality for completeness, coherence and elegance. The Macintosh, the iPod and the iPhone are well known.
- Amazon's mission statement is, "To be Earth's most customer-centric company," and they approach a project by "working backwards": first write the press release, then the FAQ.[R51]
- People use and love the web as soon as they see it. Writing for it is less yummy, though.
- Spreadsheet are loved (especially by accountants); VisiCalc is what made PCs take off.
- Porsches and Corvettes are yummy.

By contrast, Microsoft Word, Linux and the Honda Accord are good products, but not yummy.

So what? Is it important for your system to be yummy? If it's a consumer product it certainly helps a lot, and it might be crucial. For an enterprise product, staying power is more important. Clearly there's a lot of noise, but to cheaply boost your chances of making a yummy system, Amazon's approach is best. Much more expensive, but even better, is to study the users deeply.

3.6.1 *User interfaces*

And the users exclaimed with a snarl and a taunt, "It's just what we asked for but not what we want." —Anonymous[Q57]

People think that good user interfaces are all about dialog boxes, animations, pretty colors and so forth. Two things are much more important:

- The *user model* of the system: is there a way for the user to think about what the system is doing that makes sense, is faithful to what it actually does, and is easy to remember?
- *Completeness and coherence* of the interface: can the user see clearly how to get their whole job done, rather than just some piece of it? Are there generic operations like copy and paste that tell the user what operations are possible? Do the parts look and feel like a coherent design?

User models and coherence are hard because it's hard to find out what the users really need. You can't just ask them, because they are paid to do their jobs, not to explain them. No user would have asked for the iPhone. The only way is to watch them at their work or play for a long time.

Here are some examples of good user models:

- Files and folders on the desktop.
- The web, with links that you click on to navigate.
- Web search, which pretty often finds what you're looking for.
- Spreadsheets, which can do complex calculations without any notion of successive steps.

And here are some less good examples:

- Microsoft Word, with styles, sections, pages, and other things interacting confusingly.
- The user interface to security—there's no intelligible story about what's going on.

– System administration, where the sound idea that the user should describe the desired state by a few parameters is badly compromised by poor engineering of the components.

»Bravo and Gypsy. The most successful application on the Alto was the Bravo editor , the first What You See Is What You Get editor. When Charles Simonyi and I designed it, we made a deliberate decision not to work seriously on the user interface, because we knew it was hard and we didn't have the resources to both build an editing engine and invent a new UI. Larry Tesler and Tim Mott came along with their Gypsy system for the book editors at Ginn. Their first step was to spend several weeks watching their customers at their daily work. They completely replaced our UI, and they invented modeless commands and copy/paste, the basis of all modern UIs.[R47]

3.7 Incremental

There are three aspects to incremental:

– *small* steps—otherwise it wouldn't be incremental,
– *meaningful* steps—you get something useful each time, and
– steps *proportionate* to the size of the change—you don't have to start over.

Incremental can be qualitative or quantitative. Qualitative ones are being and becoming, indirection, subclassing, path names and many other techniques. Quantitative ones add elements:

– Nodes to the Internet or a LAN (and you don't even have to take it down).
– Peripherals to a computer.
– Applications to an OS installation or extensions to a browser.

3.7.1 *Being and becoming*

This is an opposition: being is a map that tells you the values of the variables, becoming a log of the actions that got you here. Some examples:

- A bitmap can represent an image directly, but so can a "display list" of drawing commands that produce the image, which generalizes to an arbitrary program, as in PostScript.
- A log-structured file system uses the log to store the data bytes, with an index just like the one in an ordinary file system except that the leaf nodes are in the log. Amazon's Aurora pushes this to a limit.
- A sequence of states, such as the frames of a video or successive versions of a file, compresses into a few complete states (called key frames for MPEG videos, checkpoints in other contexts) together with "deltas", actions that take one state to the next.
- The standard way to recover from failures in a data storage system is to apply a redo log that produces the current state from a persistent state that reflects only some prefix of the actions.
- A more general approach to fault tolerance uses a replicated state machine, which applies the same log to several identical copies of the state.

How do you find the value of a variable (that is, construct the map) from the log? Work backward through the log, asking for each logged action u how it relates to the read action r. If u is a blind write $m(a_1) := x$, and r is **return** $m(a_2)$, then either u and r commute (if $a_1 \neq a_2$) or u determines the result x of r regardless of anything earlier in the log.

»Bravo undo. How do you undo some actions to get back to a previous version v? Simply replay the log up through the last action that made v. We did this in Bravo, logging the user commands, although our original motivation was not undo but reproducing bugs, so the replay command was called `bravobug`. I've never understood why later systems didn't copy this; perhaps they didn't want to admit that they had bugs.[R33]

Optimizations

There are many variations on these ideas. To keep a log from growing indefinitely you can take a *checkpoint*, which is a map as of some point in the log. You can *share* parts that don't change

among multiple versions; a copy-on-write file system does this, as does a library for immutable data like `immutablejs`.

The idea behind these optimizations is to deconstruct the map, moving it closer to a log. The base case that the hardware provides is a fixed-size finite array of bytes in RAM, pages on disk or whatever; here the variables are integers called addresses A. Call this a store $S: A_S \to V$ and represent it abstractly by a hierarchical structure $S = A_S \to \big((T, A_T)\ \mathbf{or}\ V\big)$, where A_T is an address in a lower level store T. Each level takes an address and either produces the desired value or returns a lower level store and address. You can think of this as a way to compress a log of updates. Log structured memory is one example of this idea.

To efficiently build a store S on top of lower-level stores T_1, T_2, ..., build an index from (ranges of) S addresses $[a_S, a_S + \Delta]$ to pairs $\big(T_i, a_{T_i}\big)$; each entry in this index is a *piece*. A write changes the index for the range of addresses being written (fig. 2a). There are many data structures that can hold the index: a sorted array, a hash table, a balanced tree of some kind.

Since the T_i are stores themselves, this idea works recursively. And the indexes can be partial overlays, with a sequence of stores $S_n, S_{n-1}, \ldots S_0$; if a is undefined in $S_n, \ldots, S_i$ then you look in S_{i-1}. Several successive writes can appear explicitly or you can collapse them to a single level (fig. 2b, with just S_2 and S_0, like CPU store buffers), or all the way to an index that maps every address (fig. 2c, like a copy-on-write file system).

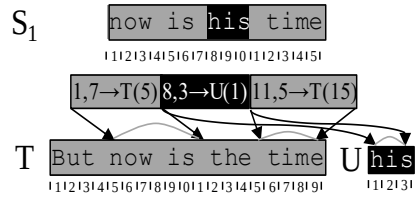


Fig. 2a: Writing "his" in place

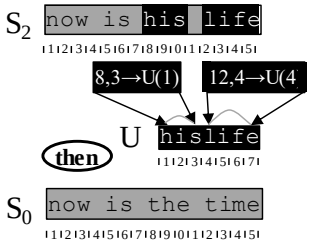


Fig. 2b: A single discontinuous write

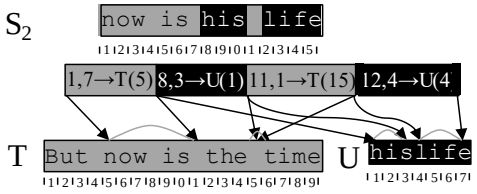


Fig. 2c: Back to a full index for S_2

Amazon Aurora applies many of these ideas to a cloud database, separating storage completely from the database code. It treats the redo records that contain database writes as the truth; when the database reads a page, storage reconstructs it from the redo records. If there are many of them, it takes a checkpoint just for that page. This drastically reduces write bandwidth.[R50]

3.7.2 *Indirection*—Take a detour, see the world.

Indirection is in opposition to inlining, but there are many other examples; a lot of them have to do with binding a client resource less tightly to the code or objects that implement it. Recall that indirection replaces the direct connection between a variable and its value, $v \to x$, with an indirect connection or link, $v \to u \to x$. This means that you go through intermediary u to get to the object, and u can do all kinds of things. It can *multiplex* x onto some bigger object or *federate* it with y so that its own identity becomes invisible. It can *encapsulate* x, giving it a different interface to make it more portable or more secure. It can *virtualize* x, giving it properties its creators never dreamt of. It can *interpose* between v and x to instrument the connection. It can act as a *name* for x, decoupling x from its clients and making it easy to switch v to a different x.

Multiplexing divides up a resource into parts. The classic example is dividing a communication channel into subchannels, either statically by time, frequency, or code division multiplexing, or dynamically by packet switching. An OS multiplexes files onto a disk or processes onto a CPU.

Routing does this repeatedly; Internet packets, email messages and web page requests all go through several indirections.

Federation is almost the opposite, combining several resources into a single one: several disks into one volume, several filesystems into a bigger one by mounting, a sea of networks into the Internet. Load-balancing federates servers: each client sees a single resource, but there are many clients and the balancer spreads the load across many servers.

Encapsulation isolates a resource from its host, as a *secure enclave* that keeps the resource safe from the host or a *sandbox* that keeps the host safe from an app.

Virtualization converts a "physical" host resource into a "logical" guest one that is less limited (virtual memory much bigger than physical memory, missing instructions trapped and done in software) and easier to move (virtual machines not bound to hardware). It can also change the interface, for example with a different ISA on the guest so you can run old programs (*emulation*) or for portability, as with the Java Virtual Machine (JVM). An interpreter can run the guest ISA by executing instructions of the host, or a compiler can translate guest programs to the host ISA either statically, or dynamically using JIT. Other examples: virtual hard disks, overlay networks, the C library. An adapter can handle a smaller interface change.

Interposing splices more or less arbitrary code between a client and a service, often to log audit records or to collect information about performance. It's easy to do this for a class, but it's always possible, even at the level of machine instructions. Proxies and content distribution networks such as Akamai do this on a larger scale to distribute load and improve locality.

Naming decouples a service such as Twitter from the physical machines that implement it. In that example there are several levels: DNS maps `twitter.com` to an IP address, and the Internet delivers packets with that address to a machine. Similarly, a style in a word processor names a group of character or paragraph properties, decoupling the markup from the final appearance, and a mailing list, security group or role names a group of people, decoupling the structure of an organization from the current membership. An index makes name lookup or search cheaper. Indirection makes it easier to have *aliasing*: several different v's that map to the same x.

Certificates use indirection to establish trust.

4 Process

The most important single aspect of software development is to be clear about what you are trying to build. —Bjarne Stroustrup[Q48]

Systems resemble the organizations that produce them (paraphrased). —Melvin Conway[Q9]

If you can't be a good example, then you'll just have to be a horrible warning. —Catherine Aird[Q1]

SOFTWARE IS HARD. … Good software … requires a longer attention span than other intellectual tasks. —Donald Knuth[Q27]

The acronym for process is ART: **A**rchitecture, **A**utomation, **R**eview, **T**echniques, **T**esting. I don't have much personal experience with this. But I have watched a lot of systems being developed, with teams that range in size from six to several thousand people. If you find yourself working on a team that breaks the rules in this section, it's time to find another job.

You can build a small system with willpower: one person keeps the whole design in their head and controls all the changes. You can even do without a spec. But a system that's bigger (or lives for a long time) needs process. Otherwise it's broken code and broken schedules. Process means:

- Architecture: Design that gets done, and documented so that everyone can know about it.
- Automation: Code analysis tools (very cheap for the errors they can catch) and build tools.

- Review: Design review—manual, but a much cheaper way to catch errors than testing.
- Review: Code review—manual, but still cheaper than testing.
- Testing: Unit and component tests; stress and performance tests; end-to-end scenarios.[R9]

None of this will help, though, if the goal is badly conceived. If your system isn't going to be yummy, it had better at least be useful. If it's entering a crowded field, it needs to be a *lot* better than the market leaders. If there's a strong ecosystem of languages and applications in place, build on it rather than fighting it. And usually simplicity is key: if your system does one thing well, it's easier to sell and easier to build. If it's successful it will expand later. Some widely known examples:

- Dropbox just syncs a subtree of the file system.
- The C language stays as close to the machine as possible.
- HTML (the original) gives you links, text with simple formatting, and bitmap images.
- Twitter gives you 140-character tweets that can go to millions of followers.

The symbiotic relationship between a platform and its applications can take one of two forms:

- **Controlled**: The platform only accepts applications that fit its self-image, with the goal of coherence and predictability for the whole ecosystem. Apple does it this way.
- **Wild and free**: The platform accepts anything, and it's up to the market to provide whatever coherence there is. Windows does it this way. Android is in the middle.

»Intel Itanium. When Intel made a big bet on a VLIW (Very Long Instruction Word) design for its 64 bit Itanium architecture to replace the x86, the performance predictions were apparently based on a single hand-coded inner loop, 30 instructions long, since they didn't have the optimizing compiler working.[R15] Most real programs turned out to be less amenable. Usually chip designs are based on extensive simulation of real workloads.

5 Oppositions

Finally, here is a brief exploration of each opposition.

Simple ↔ rich, fine ↔ features, general ↔ specialized [S Y]

KISS: Keep It Simple, Stupid. Do one thing well. Don't generalize.
Don't hide power. Leave it to the client. Make it fast. Use brute force.

If in doubt, leave it out. —Anonymous

The cost of adding a feature isn't just the time it takes to code it, [it's the] obstacle to future expansion. ... Pick the features that don't fight each other. —John Carmack[Q6]

Systems are complicated because it's hard work to make them simple, and because people want them to do many different things. You can read a lot about software bloat, the proliferation of features in browsers and in rich applications like Word and Excel. But each of those features has hundreds of thousands of users at least. The tension between keeping things simple and doing a lot is real, and there is no single right answer, especially for applications that interact with users.

Still, it's best to add features and generality slowly, because:

- You're assuming that you know the customers' long-term needs, and you're probably wrong. It's hard enough to learn and meet their immediate needs.
- It takes time to get it right, but once it's shipped legacy customers make it hard to change.
- More features mean more to test, and more for a bad guy to attack.

So why do systems get overambitious? Because there are no clear boundaries,[Q5] as there are with bridges for example, and programmers are creative and eager to tackle the next challenge. But features that have a lot in common can add power without adding too much complexity; it's best

to do this with a single mechanism that takes different parameters for the different features. So a search engine can index many different data types, a webpage can include text, images and video, or an email program can keep a calendar.

For software whose clients are other programs, the solution is building programs on components. A single component should do one thing, and its code should do it well and predictably so that clients can confidently treat it as a primitive building block; beware of components that don't have these properties. With a good set of such components a client can do a lot without writing much code, relying on them to take care of most performance issues. Some examples: key-value stores; Unix shell programming on top of primitives like `diff`, `sort`, `grep`; mathematics systems like Mathematica and Julia. Building one of these components is a lot of work. It's worth doing if the component is critical for your system, or if it's part of a platform like an operating system or a browser where it will have lots of clients.

Perfect ↔ adequate, exact ↔ tolerant [S T D] —Just good enough. Flaky, springy parts.

Worse is better. —Richard Gabriel[Q18]
The best is the enemy of the good. —Voltaire[Q51]

This is not about whether there is a precise spec, but about how close the answer needs to be to an ideal result. "Close" can take different forms: a tolerance or a probability of being right, results that may just be wrong in some difficult cases, or a system that behaves well as long as its environment does. Some examples:

Tolerance or **probability**:

- Available 99.5% of the time (down no more than one hour per week), rather than 100%.
- Response time less than 200 ms with 99% probability, rather than always.
- A 98% hit rate in the cache on the Spec benchmark, rather than 100%.

Such properties usually come from a randomized algorithm, or as statistics derived from measuring a running system.

Wrong in difficult cases:

- Words are hyphenated if they appear in a hyphenation dictionary, rather than always.
- Changes to DNS may not appear immediately, because it uses eventual consistency.
- A database system may fail, but it recovers without losing any committed work.

Friendly environment. Every system at least depends on its host to execute its instructions correctly, but often the system can be simpler or cheaper by assuming more about its environment:

- Data is not lost as long as the power doesn't fail.
- Your files are available if you have a connection to the Internet.
- Faces are recognized reliably if the lighting is good enough.

The environment is not just the host you depend on; it's also your clients. If they are not too demanding, your system may be adequate even if it doesn't satisfy an ideal spec.

Spec ↔ code [S]

Keep secrets. Good fences make good neighbors. Free the implementer.
Embrace nondeterminism. Abstractions are leaky.

Don't tie the hands of the implementer. —Martin Rinard[Q42]
Writing is nature's way of letting you know how sloppy your thinking is. —Richard Guindon[Q19]

A spec tells you *what* a system is supposed to do, and the code tells you *how*. Both are described by actions; how do they differ? A spec constrains the visible behavior of the system by saying what behaviors (sequences of steps) are acceptable or required. A spec is not a program, and the right language for writing it is either English (if it's not time to be precise) or mathematics.

The code is executable, but it still may not be a program you can run; it may be an algorithm such as Quicksort or Paxos, described precisely in pseudocode that abstracts from the details of how the machine represents and acts on data.

Declarative ↔ functional ↔ imperative [S E] —Say what you want. Make it atomic.

The many styles of programming can be grouped into three broad classes: declarative, functional and imperative.

An imperative program (for example, one written in Java or C) has a sequence of steps and a program counter, as well as named variables that the program can read or write. Interesting programs take lots of steps thanks to loops or recursion. Most computing hardware is imperative.

A functional program (perhaps written in the functional subset of Haskell) has function calls instead of steps, and immutable values bound to function parameters or returned from the calls instead of state variables. Interesting programs have recursive functions, so they can make lots of calls. Real languages aren't purely functional because small changes to big values are too expensive, but you can embed immutable data structures in an imperative language, and a library like `immutablejs` can make this efficient. The most widely used programming languages are functional: spreadsheets and database query systems. However, they are special-purpose.

The literature doesn't say what a declarative program is, but I think it's a program with few steps; people are not very good at understanding long sequences of steps. Often it's also easier to optimize, since it doesn't commit to the sequence of steps the machine should take. Powerful primitives help to make a program declarative; for example, code to compute a transitive closure has lots of steps, but a transitive closure primitive is a single easy step. The SQL query language for relational databases has many such primitives, as does HTML as an abstract description of a desired webpage.

Precise ↔ approximate software [T D] —Get it right. Make it cool. Shipping is a feature.[Q36]

Unless in communicating with [a computer] one says exactly what one means, trouble is bound to result. —Alan Turing[Q50]

Vaguely right is better than precisely wrong. —Leonard Lodish[Q30]

Broadly speaking, there are two kinds of software, precise and approximate, with the contrasting goals "Get it right" and "Get it soon and make it cool".

Precise software has a specification, even if it's not written down very precisely, and the customer is unhappy if the software doesn't satisfy its spec. Obviously software for controlling airplanes or nuclear reactors is precise, but so are word processors, spreadsheets, and software for handling money. The spec might be nondeterministic, but that doesn't mean that it's imprecise.

Approximate software, on the other hand, has a very loose spec, or none at all; the slogan is "Good enough." The packet-switched Internet, web search, retail shopping, face recognition, and social media are approximate.

Approximate software is not better or worse than precise, but they are very different, and it's important to know which kind you are writing. If you wrongly think it's precise, you'll do extra

work that the customers won't value, and it will take too long. If you wrongly think it's approximate, the customers will be angry when code doesn't satisfy the (unwritten) spec they counted on.

Dynamic ↔ static [E A] —Stay loose. Pin it down. Shed load. Split resources.

A computer is infinitely flexible, but a program is not; both what it does (the spec) and how (the code) are more specialized. Yet the code can be more or less able to adapt to changes in itself or in the environment. Flexibility is costly; code that takes advantage of things that stay constant is more efficient, and static checking automatically proves theorems about your code before you ship it. To some extent you can have both with *just-in-time* (JIT): make a static system based on the current code and environment, and remake it if there are changes.

There are (at least) four aspects of this opposition: interpret vs. compile, indirect vs. in-line, scalable vs. fixed, and online vs. preplanned resource allocation.

Compiling commits the code to running on a host that is usually less flexible and closer to the hardware. The compiler chooses how data is represented, and often it infers properties of the code (example: at this point $v = 3$ always) and uses them to optimize. It may do *trace scheduling*, using information from past runs or heuristics to predict code properties (in this JavaScript program, i is usually an integer).[R22] These predictions must be treated as hints and checked at runtime, with fallback to slower code when they are wrong. Together with JIT, trace scheduling can adapt a very general program to run efficiently in common cases.

A different aspect of the dynamic-static opposition is resource allocation, and scheduling in particular. CPUs and distributed systems can allocate resources online to a sequence of tasks that's not known in advance (using caches, branch prediction, asynchronous concurrency, etc.), but if you know the sequence you can do this work just once. Example: resources reserved for a real-time application, anda *systolic* array in which work items pass through a sequence of processors with no queuing.[R27] Storage allocation is similar; static allocation (splitting up the storage) is cheaper if you know the sizes in advance or can guess them well. And when it fails, it's much easier to figure out why.

Indirect ↔ inline [E I] —Take a detour, see the world.

Any problem in computing can be solved by another level of indirection. —David Wheeler[Q52]
Any performance problem can be solved by removing a level of indirection. —M. Haertel[Q20]

Indirection is a special case of abstraction that replaces the direct connection between a variable and its value, $v \to x$, with an *indirect* connection $v \to u \to x$, often called a *link*; the idea is that ordinary lookups to find the value of v don't see u, so that clients of v don't see the indirection. You can change the value of v by changing u, without changing x. Often u is some sort of service, for example the code of a function, reached indirectly by jumping to the code for the function; this gives the most flexibility, since you can run arbitrary code in the service. The link doesn't have to be explicit; it could be an *overlay* that maps only some of the possible v's, like a TLB or a cache.

Inlining replaces a variable v with its value x. This saves the cost of looking up v, and the code can exploit knowing x. For example, if $x = 3$ then $x + 1 = 4$; this saves an addition at runtime. If v is a function you can inline its code, avoiding the control transfer and argument passing, and now you can specialize to this particular argument. But inlining takes more space and makes it hard to change the function's code.

Lazy ↔ eager ↔ speculative [E] —Put it off. Take a flyer.

When you come to a fork in the road, take it. —Fort Gibson New Era[Q56]

The common theme is to improve efficiency by reordering work. The base case is *eager* execution, which does work just when the sequential flow of the program demands it; this is the simplest to program. **Lazy** execution *defers* work until it must be done to produce an output, gambling that it will never be needed. It can pay off in lower latency because it first does the work that produces output, and in less work if the output turns out not to be needed at all.

Indirection is lazy as well as dynamic—if you never need the value of the name, you never pay the cost of following the link. Other examples are write buffers, which defer writes from a cache to its backing store; redo logging, which replays the log only after a crash; eventual consistency, which applies updates lazily and in an arbitrary order until there's a need for a consistent result.

More generally, it's lazy to represent a function by code rather than as a set of ordered pairs. Of course if the set is infinite then code is the only option. Pushing this idea farther, to defer the execution of some code, wrap it in a function and don't invoke it until the result is needed.

Speculative execution *anticipates* work in advance, gambling that it will be useful. This makes sense if you have resources that are otherwise idle, or to reduce latency in the future. Prediction is the most common form of speculation, for example when a storage system prefetches data from memory to cache, or when a CPU predicts which way a branch instruction will go. Caching speculates that an entry will be used before it has to be replaced. Exponential backoff in networks and optimistic concurrency control in databases speculate that there will be little contention.

Usually laziness or speculation keeps the program's results unchanged. This is simplest if the parts being reordered commute. They do in a functional program, but code with side effects may not. Sometimes, as with eventual consistency, you settle for sloppy results.

Centralized ↔ distributed, share ↔ copy [E D] —Do it again. Do it twice. Find consensus.

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. —Leslie Lamport[Q29]

If you have a choice, it's better to be centralized. Distributed systems are more complicated because they have inherent concurrency and partial failures, and they have to pay for communication. But they are essential for serious fault tolerance, and for scaling beyond what you can get in a single box. A distributed system needs fault tolerance because it has to deal with *partial failures*; you don't want to crash the whole system when one component fails. But even a very large system can be centrally managed (in a fault-tolerant way) because management doesn't require that much computing or data; this is how large cloud systems like AWS and Azure work.

Fixed ↔ evolving, monolithic ↔ extensible [A I]

The only constant is change. Make it extensible. Flaky, springy parts.

No matter how far down the wrong road you have gone, turn back now. —Turkish proverb

Always design your program as a member of a whole family of programs, including those that are likely to succeed it. —Edsger Dijkstra[Q13]

It's cheaper to replace software than to change it. —Phil Neches[Q35]

Often the customer's needs are unclear, and successful systems tend to live for a long time, during which the needs change. Just thinking hard is usually not enough to make unclear needs clear,

because you aren't smart enough. It's better to follow the agile model: build a prototype, try it out, and improve it.[R21]

A successful system must do more—it must evolve, because needs change as people see ways to make it do more, as the number of users grows, as the underlying technology changes, and as it interoperates with other systems that perhaps didn't even exist originally. Evolution requires modularity, so that you can change parts of the system without having to rebuild it completely. Interfaces allow clients and code to evolve independently. These are aspects of divide and conquer.

Evolution is easier with *extensibility*, a well-defined way to add certain kinds of functionality. This is a special form of modularity, and it needs a lot of care to keep from exposing secrets of the code that you might want to change. Examples:

- You can add new kinds of tags to HTML, even very complicated ones, and old implementations will simply ignore them.
- Most operating systems can incorporate any number of I/O drivers that know about the details of a particular scanner, printer, disk, or network.
- Inheritance in programming languages like Smalltalk and Python makes it convenient (if dangerous) to add functionality to an existing abstraction.

Another way to extend a component is to let the client pass in a (suitably constrained) program as an argument; for example, a search engine can take a parser for an unfamiliar format. You can do this without pre-planning by patching, but it's tricky to maintain all the code's invariants.

Policy ↔ mechanism [A] —It's OK to change your mind.

When the facts change, I change my mind. What do you do, sir? —Paul Samuelson[Q44]

The mechanism is what the system *can* do, determined by its specs and code, and the policy is what the system *should* do: the control system for the mechanism. Policy is different for each installation, typically changes much faster than the code, and is set by administrators rather than developers. It should give them as much control over the mechanism as possible.

The most elaborate example of the distinction is in security, where the mechanism is access control and the policy is what principals should have access to what resources. Other examples: policy establishes quotas, says how much replication there should be, or decides what software updates should be applied. Policy is an aspect of system configuration, which also includes the hardware and software elements that make up the system and the way they are interconnected. Historically all these things were managed by hand, but cloud computing has forced automation.

Consistent ↔ available ↔ partition-tolerant [D] —Safety first. Always ready. Good enough.

If you want a system to be *consistent* (that is, all the parts of it see the same state) and highly *available* (very unlikely to fail, because it's replicated in different places), then the replicas need to communicate. But if the replicas are *partitioned* then they can't communicate. So you can't have all three; this is the CAP "theorem". The way to get around it in practice is to make partitioning very unlikely. A partial mitigation is *leases*, which are locks that time out, using the passage of real time for uninterruptible communication.

Generate ↔ check —Trust but verify.

A problem is in complexity class NP if finding a solution is hard (takes work $O(2^n)$), but checking it is easy (work $O(n^k)$). The most common place for a check is in an `assert`, but there are many others such as proof-carrying code. The general idea, however, is much broader: keep a hint that

might be wrong, but is easy to check. This is a narrower meaning of "hints" than in the title of this paper. The end-to-end principle is closely related.

Being ↔ becoming [I] — How did we get here? Don't copy, share.

There are two ways to represent the state of a system:

- *Being*: the values of the variables—a *map* $v \to x$
- *Becoming*: a sequence of actions that gets the state to where it is—a *log* of actions.

Different operations are efficient in different representations. If you're only interested in a single point in time, you want the map. If you care about several different versions (to recover the current state from a checkpoint, undo some actions, or merge several versions), you want the log. There are ways to convert one representation into the other, and points between the extremes: applying the actions gets you the values, a `diff` produces a *delta* (a sequence of actions that gets you from one state to another), *checkpoints* shorten the log. Ordinary programs use being; fault-tolerant programs use both. More on this here.

Iterative ↔ recursive, array ↔ tree [I] —Treat the part like the whole.

To iterate is human, to recurse divine. —Peter Deutsch[Q11]

There are few things known about systems design, but the basic principle of recursive design is: make the parts of the same power as the whole. —Bob Barton[Q3]

Iteration and recursion are both Turing-complete. You can write an iteration recursively using tail-recursion (which is easy: the last step in the loop is the only recursive call), and you can write a recursion iteratively using a data structure to simulate a call stack (which is a pain).But iteration is more natural when there's a list or array of unstructured items to process, and recursion is more natural when the items have subparts, especially when the parts can be as general as the whole.

Thus recursion is what you want to process a tree or a graph where the description of the structure is itself recursive. Here are examples that illustrate both points:

- A hierarchical file system can have different code at each directory node. Some nodes can be local, others on the Internet, yet others the result of a search: `bwl/docs/?author=smith`.[R23]
- Internet routing is hierarchical, using BGP at the highest level and other protocols within an Autonomous System.

These examples also show how a *path name* (a sequence of simple names) identifies a path in a graph with labeled edges and provides decentralized naming. Just as any tree node can be the root of an entire subtree, a path name can grow longer without conflicting with any other names.

6 Conclusion

I don't know how to sum up this paper briefly, but here are the most important points:

- Keep it simple. Complexity kills.
- Write a spec. At least, write down the abstract state.
- The ABCs of efficiency: algorithms, approximate, batch, cache, concurrent (shard, stream).
- Being vs. becoming: map vs. log, pieces, checkpoints, indexes.
- Eventual consistency: local data, high availability, sharding.

Quotes

I've tried to find attributions for all the quotations; some were unexpected, and it's disappointing that some of the best ones seem to be apocryphal. References of the form [Author99] are to PDF files that might not be at the link I've given. You'll find them here.

Q1. Catherine Aird, *His Burial Too*, Collins, 1973.

Q2. Dan Ariely, You are what you measure, *Harvard Business Review* **88**, 6, June 2010, pp 38-41. Link [Ariely10]

Q3. Bob Barton, quoted by Alan Kay in The Early History of Smalltalk, *ACM Conf. History of Programming Languages II*, *SIGPLAN Notices* **28**, 3, March 1993, pp 69-95.

Q4. Yogi Berra, *Inspiration and Wisdom from One of Baseball's Greatest Heroes*, Hyperion, 2002, p. 53.

Q5. Fred Brooks, No silver bullet, *IEEE Computer* **20**, 4 (April 1987), pp 10-19. Link [Brooks87]

Q6. John Carmack, Archive - .plan (1997), July 7, 1997, p 41. Link [Carmack97]

Q7. General Benjamin W. Chidlaw, Commander in Chief, Continental Air Defense Command, 1954. Link

Q8. John Robert Colombo, A Said Poem, in *Neo Poems*, The Sono Nis Press, Department of Creative Writing, University of British Columbia, 1970, p 46. Attributed to Mark Twain without evidence by Colombo and many others. Link [Colombo70]

Q9. Melvin Conway, How do committees invent?, *Datamation*, **14**, 5, April 1968, 28–31. The original is, "Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations." Link [Conway68]

Q10. Terry Crowley, What to do when things get complicated, *Hacker Noon*, Sep. 27, 2017. Link [Crowley17-9-27]

Q11. Peter Deutsch, quoted in James O. Coplien, *C++ Report* **10** (7), July/August 1998, pp 43-51. Sometimes attributed to Robert Heller. Link. Also quoted in Bjarne Stroustrup, *The C++ Programming Language*, Special Edition (3rd Edition), Addison-Wesley, 2000, ch. 7, p 143.

Q12. Philip K. Dick, How to build a universe that doesn't fall apart two days later. In *The Shifting Realities of Philip K. Dick*, Vintage, 1995. Link [Dick95]

Q13. Edsger Dijkstra, quoted in *In Pursuit of Simplicity*, University of Texas, Austin, May 2000. Link

Q14. Edsger Dijkstra, The humble programmer, *Comm. ACM* **15**, 10, Oct. 1972, pp 859-866. Link

Q15. Edsger Dijkstra, My hopes for computing science, *Proc. 4th International Conference on Software Engineering* (ICSE '79), Munich, 1979, pp 442-448. Link, Link [EWD709] [Dijkstra79].

Q16. Albert Einstein, On the Method of Theoretical Physics, the Herbert Spencer Lecture, Oxford, June 10, 1933, *Philosophy of Science* **1**, 2, April 1934, p 165. Einstein actually said, "It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience." Link [Einstein33]. Roger Sessions gave the concise version: "I also remember a remark of Albert Einstein …. He said, in effect, that everything should be as simple as it can be but not simpler!" in How a 'Difficult' Composer Gets That Way, *New York Times*, January 8, 1950. Link [Sessions50]

Q17. Bob Frankston, in Martha Baer, Immortal code, *Wired*, February 1, 2003. Link

Q18. Richard Gabriel, Worse is better. Link [Gabriel91]

Q19. Richard Guindon, *Michigan So Far*, Detroit Free Press, 1991, p 110.

Q20. M. Haertel, from Link.

Q21. Tony Hoare, The emperor's old clothes, *Comm. ACM* **24**, 2, Feb. 1981, pp 75-83. Link

Q22. Samuel Johnson, in James Boswell, *Life of Johnson*, John Sharpe, London, 1830, p 540 (April 18, 1783). The original is, "JOHNSON. 'Were I a country gentleman, I should not be very hospitable, I should not have crowds in my house.' BOSWELL. 'Sir Alexander Dick tells me, that he remembers having a thousand people in a year to dine at his house: that is, reckoning each person as one, each time that he dined there.' JOHNSON. 'That, Sir, is about three a day.' BOSWELL. 'How your statement lessens the idea.' JOHNSON. 'That, Sir, is the good of counting. It brings every thing to a certainty, which before floated in the mind indefinitely.' BOSWELL. 'But Omne ignotum pro magnifico est: one is sorry to have this diminished.' JOHNSON. 'Sir, you should not allow yourself to be delighted with errour.' BOSWELL. 'Three a day seem but few.' JOHNSON. 'Nay, Sir, he who entertains three a day, does very liberally.'"

Q23. Juvenal, Satire 6, ll 346–348. "Quis custodiet ipsos custodes? Qui nunc lasciuae furta puellae hac mercede silent." Link

Q24. Alan Kay, quoted by Andy Hertzfeld in *Creative Think*, 1982. This is the best citation I could find, but surely it's much older. Link [Hertzfeld82]

Q25. Alan Kay, who says that this is "a saying I made up at PARC," in Brian Merchant, The Father of Mobile Computing is Not Impressed, *Fast Company*, Sep. 15, 2017. Link [Kay17]

Q26. Alan Kay, in *InfoWorld*, June 11, 1984, p 59. Link [Infoworld84]. But he says here that this quote is from *Newsweek*, 1984.

Q27. Donald Knuth, *Selected Papers on Computer Science*, Stanford: Center for the Study of Language and Information, 1996, p 161.

Q28. Leslie Lamport, *Bulletin of EATCS* 125, June 2018, pp 96-116. Link [Lamport18]

Q29. Leslie Lamport, Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 1987. Link No. 75, Link

Q30. Leonard Lodish, 'Vaguely right' approach to sales force allocations, *Harvard Business Review* **52**, 1, January-February 1974, pp 119-124. [Lodish74]

Q31. Somerset Maugham, quoted without citation in Ralph Daigh, *Maybe You Should Write a Book*, Prentice-Hall, 1977. Link

Q32. Gary McGraw, Software Assurance for Security, *IEEE Computer* **32**, 4, April 1999, pp 103-105. Link [McGraw99]

Q33. Joni Mitchell, Both sides now, on *Clouds*, Reprise Records, May 1969. Link [Mitchell67]

Q34. Don Mitchell, Brainstorming–Its application to creative advertising, *Proc. 13th Annual Advertising and Sales Promotion Executive Conference*, Ohio State University, October 26, 1956, p 19. Misattributed to Walt Kelly in the form, "We are faced with an insurmountable opportunity." Link

Q35. Phil Neches, who told me in August 2019 that he did say this, but did not know where it was published. I have not been able to find a citation either. I saw it in a list of four such quotes attributed to Neches, which I now can't find either.

Q36. Mike Neil, *Viridian features update; beta planned for Longhorn RTM*, Windows Server Blog, May 10, 2007. Link [Neil07]

Q37. William of Occam. This formulation, "Entia non sunt multiplicanda praeter necessitatem," is due to the Irish Franciscan philosopher John Punch in his 1639 commentary on the works of Duns Scotus. Link

Q38. John Ousterhout, who told me in August 2019 that he hasn't published this anywhere. He goes on to say, "I've found that in most situations the simplest code is also the fastest. … Tune only the places where you have measured that there is an issue." Link [Ousterhout19]

Q39. Blaise Pascal, *Lettres provinciales*, letter 16, 1657. « Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte. » Misattributed to Mark Twain. Link for the history. Link for the original.

Q40. President's Council of Advisors on Science and Technology, *Designing a digital future: Federally funded research and development in networking and information technology*, Technical report, Executive Office of the President, 2010, p 71. Link [PCAST10]

Q41. Rob Pike, 5 rules of programming, rule 3, in "Notes on programming in C," February 21, 1989. Link [Pike89]

Q42. Martin Rinard, in MIT course 6.826, 2002.

Q43. Larry Rudolph suggested this term.

Q44. Paul Samuelson, *Meet the Press*, December 20, 1970, transcript published in the Daily Labor Report 246, Bureau of National Affairs Inc., December 21, 1970, p X-3. Misattributed to Keynes, though in 1978 Samuelson did attribute it to him. Link.

Q45. Bruce Schneier, *Dr. Dobb's Journal*, Dec. 2000. Link, Link [Schneier00]. Also in Bruce Schneier, *Secrets and Lies*, ch. 23, Wiley, 2000.

Q46. Seward, *Biographiana*, footnote to entry for Abbé Marolles. Apparently there is no direct citation. Link.

Q47. Joel Spolsky, Things you should never do, part I, April 6, 2000. Link [Spolsky00-4-6]

Q48. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1997, p 692.

Q49. John Tukey, The future of data analysis. *Annals of Mathematical Statistics* **33**, 1, 1962, p 13. Link [Tukey62]

Q50. Alan Turing, Lecture on the automatic computing engine, Feb 20, 1947, in *The Essential Turing*, Oxford, 2004, p 392. Link. [Turing47, p 12]

Q51. Voltaire, 'La Bégueule', *Contes*, 1772, l 2. Link. « Dans ses écrit un sage Italien / Dit que le mieux est l'ennemi du bien. » He credits the Italian proverb in *Dictionnaire philosophique*, 1764, Art dramatique, Du récitatif de Lulli. Link. For the Italian, see Pescetti, *Proverbi Italiani*, 1603, p 30. Link

Q52. David Wheeler, but I don't know a direct citation for this (it has been widely but wrongly attributed to me). It appears in Bjarne Stroustrup, *The C++ Programming Language* (4th Edition), Addison-Wesley, 2013, p v. Link [Stroustrup13]. Stroustrup was Wheeler's PhD student. But "It is easier to move a problem around (for

example, by moving the problem to a different part of the overall network architecture) than it is to solve it." In RFC 1925.

Q53. A.N. Whitehead, *An Introduction to Mathematics*, Holt, 1911, ch. 5, p 43. Link

Q54. Walt Whitman, *Song of Myself*, part 51. Link

Q55. Frank Zappa, "Packard Goose", *Joe's Garage*, Act III, track 2, Zappa Records, 1979. Link

Q56. Fort Gibson New Era, *Wise Directions* (filler item), p 2, col 6, July 31, 1913, Fort Gibson, OK. The original adds, "I will, if it's a silver one." Misattributed to Yogi Berra. Link

Q57. From "'Twas the night before release date," in many places on the Internet.

References

The references include links to the ACM Digital Library where I could find them. If the Library doesn't have a PDF for an item, there's a citation of the form [Author99], and at this link there's a PDF file whose name starts with Author99.

R1. Martín Abadi and Leslie Lamport, The existence of refinement mappings, *Theoretical Computer Science* **82**, 2, May 1991, pp 253-284. Link [Abadi91]

R2. Lada Adamic, Zipf, Power-laws, and Pareto—A Ranking Tutorial, 2002. Link [Adamic02]

R3. Keith Adams and Ole Agesen, A comparison of software and hardware techniques for x86 virtualization, *Proc.12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS XII), *ACM SIGOPS Operating Systems Review* **40**, 5, Dec. 2006, pp 2-13. Link

R4. Sarita Adve and Kourosh Gharachorloo, Shared memory consistency models: A tutorial, *IEEE Computer* **29**, 12, Dec. 1996, pp 66-76. Link [Adve96]

R5. Nadav Amit et al, Bare-metal performance for virtual machines with exitless interrupts, *Comm. ACM* **59**, 1, Jan. 2016, pp 108-116. Link

R6. Andrea Arpaci-Dusseau et al, High-performance sorting on networks of workstations, *Proc. 1997 ACM Int'l Conf. Management of Data* (SIGMOD '97), *ACM SIGMOD Record*, **26**. 2, June 1997, pp 243-254. Link

R7. Jon Bentley, Don Knuth and Doug McIlroy, Programming pearls: A literate program, *Comm. ACM* **29**, 6, June 1986, pp 471-483. Link

R8. Inquiry Board, *Ariane 5, flight 501 failure*, European Space Agency, 1996. This report is a model of clarity and conciseness. Link [Ariane96].

R9. Eric Brechner, Nailing the nominals, *Hard Code*, October 1, 2008. Link [Hardcode08-10-1]

R10. Eric Brewer, Spanner, TrueTime & the CAP theorem, Feb. 14, 2017. Link [Brewer17]

R11. Andrei Broder, Identifying and filtering near-duplicate documents, *Proc.11th Ann. Symp. Combinatorial Pattern Matching* (COM '00), LNCS 1848, Springer, June 2000, pp 1-10. Link [Broder00]

R12. Dah-Ming Chiu and Raj Jain, Analysis of increase and decrease algorithms for congestion avoidance in computer networks, *Computer Networks and ISDN Systems* **17**, 1, June 1989, 1–14. Link [Chiu89]

R13. Austin Clements et al, The scalable commutativity rule: Designing scalable software for multicore processors, *ACM Trans. Computer Systems* (TOCS) **32**, 4, Jan. 2015, article 10. Link

R14. Robert Colwell, *The Pentium Chronicles*, Wiley, 2005.

R15. Robert Colwell and Paul Edwards, *Oral history of Robert P. Colwell*, ACM SigMicro, 2009, p 86. Link

R16. Terry Crowley, What to do when things get complicated, *Hacker Noon*, Sep. 27, 2017. Link [Crowley17-9-27]

R17. Jeffrey Dean and Luiz Barroso, The tail at scale, *Comm. ACM* **56** 2, Feb. 2013, pp 74-80. Link

R18. Peter Deutsch and Chuck Grant, A flexible measurement tool for software systems. *Proc. IFIP Congress* 1971, North-Holland, pp 320-326. [Deutsch71]

R19. Dawson Engler et al, A few billion lines of code later: Using static analysis to find bugs in the real world, *Comm. ACM* **53**, 2, Feb. 2010, pp 66-75. Link

R20. Agner Fog, The microarchitecture of Intel, AMD and VIA CPUs, 2018. Link [Fog18]

R21. Armando Fox and David Patterson, *Engineering Long-Lasting Software*, Strawberry Canyon, 2012. Link

R22. Michael Franz et al, Trace-based just-in-time type specialization for dynamic languages, *Proc. 30th ACM Conf. Programming Language Design and Implementation* (PLDI '09), *ACM SIGPLAN Notices* **44**, 6, June 2009, pp 465-478. Link

R23. David Gifford et al, Semantic file systems, *Proc. 13th ACM Symp. Operating Systems Principles* (SOSP '91), *ACM Operating Systems Review* **25**, 5, Oct. 1991, pp 16-25. Link

R24. Jim Gray, *Why do computers stop and what can be done about it*, Tandem Technical Report TR 85.7, 1985, p 11. Link [Gray85]

R25. Robert H'obbes' Zakon, *Hobbes' Internet Timeline 25*. Link [Hobbes18]

R26. Alex Kogan and Erez Petrank, A methodology for creating fast wait-free data structures, *Proc. 17th ACM Symp. Principles and Practice of Parallel Programming* (PPoPP '12), *ACM SIGPLAN Notices* **47**, 8, Aug. 2012, pp 141-150. Link

R27. H.T. Kung, Why systolic architectures?, *IEEE Computer* **15**, 1, Jan. 1982, pp 37-46. Link [Kung82]

R28. Leslie Lamport, *Specifying Systems*, Addison-Wesley, 2002. Link [Lamport02]

R29. Butler Lampson, Hints for computer system design, *Proc. 9th ACM Symp. Operating Systems Principles* (SOSP '83), *ACM SIGOPS Operating Systems Review* **17**, 5, Oct. 1983, pp 33-48. Link. Reprinted in *IEEE Software* **1**, 1 Jan. 1984, pp 11-28. Link
R30. Butler Lampson, Software components: Only the giants survive, *Computer Systems: Theory, Technology, and Applications*, ed. K. Sparck-Jones and A. Herbert, Springer, 2004, pp 137-146. Link [Lampson04]
R31. Butler Lampson, Practical principles for computer security, *Software System Reliability and Security*, Marktoberdorf Summer School, August 2006. *NATO Security through Science Series - D: Information and Communication Security* **9**, ed. Broy, Grünbauer and Hoare, IOS Press, 2007, ISBN 978-1-58603-731-4, pp 151-195. Link, Link [Lampson06]
R32. Butler Lampson, Lecture notes for MIT 6.826, Principles of Computer Systems, 2009. Link [Lampson09]
R33. Butler Lampson, *Alto Users Handbook*, Sep. 1979, p 54. Link
R34. Charles Leiserson et al, There's plenty of room at the top, preprint, Feb. 2019. [Moore19]
R35. Paul McKenney and John Slingwine. Read-Copy-Update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*, Oct. 1998, pp 509-518. Link [McKenney98]
R36. Michael Mitzenmacher, Compressed Bloom filters, *IEEE/ACM Trans. Networking* (TON) **10**, 5, Oct. 2002, pp 604-612. Link
R37. Theodore Myer and Ivan Sutherland, On the design of display processors, *Comm. ACM* **11**, 6, June 1968, pp 410-414. Link
R38. Chris Newcombe et al, How Amazon Web Services uses formal methods, *Comm. ACM* **58**, 4, April 2015, pp 66-73. Link
R39. O'Reilly Foo Camp (East), Microsoft New England R&D Center, May 2, 2010.
R40. Kay Ousterhout et al, Sparrow: Distributed, low latency scheduling, *Proc. 24th ACM Symp. Operating Systems Principles* (SOSP '13), 2013, pp 69-84. Link
R41. Benjamin Pierce, Types considered harmful, invited talk at *24th Conf. Mathematical Foundations of Programming Semantics* (MFPS XXIV), May 2008. Link [Pierce08]
R42. Marshall Rose, The future of OSI: A modest prediction, *Proc. IFIP TC6/WG6.5 Int'l Conf. on Upper Layer Protocols, Architectures and Applications* (ULPAA '92), North-Holland, 1992, pp 367-376. Link. Reprinted in Marshall Rose, *The Internet Message: Closing the Book with Electronic Mail*, Prentice-Hall, 1993, sec. C.2.2, p 324. [Rose92]
R43. Jerry Saltzer et al, End-to-end arguments in system design, *ACM Trans. Computer Systems* (TOCS) **2**, 4, Nov. 1984, pp 277-288. Link
R44. Nir Shavit, Data structures in the multicore age, *Comm. ACM* **54**, 3, Mar. 2011, pp 76-84. Link
R45. Joel Spolsky, Things you should never do, part I, *Joel on Software*, April 6, 2000. Link [Spolsky00-4-6]
R46. Amitabh Srivastava and Alan Eustace, Atom: A system for building customized program analysis tools, *Proc. 15th ACM Conf. Programming Language Design and Implementation* (PLDI '94), *ACM SIGPLAN Notices* **29,** 6, June 1994, pp 196-205. Link. Reprinted with a retrospective in *20 Years of PLDI*, 2003, *ACM SIGPLAN Notices* **39**, 4, April 2004, pp 528-539. Link
R47. Larry Tesler and Tim Mott, *Gypsy—The Ginn Typescript System*, Xerox, 1975. Link [Tesler75]
R48. Chuck Thacker et al, Firefly: A multiprocessor workstation, *IEEE Trans. Computers* **37**, 8, Aug. 1988, pp 909-920. Link, Link [Thacker88]
R49. New York Times, April 14, 1994, F.A.A. Is Threatening to Cancel New Air Traffic System. Link [FAA94]
R50. Alexandre Verbitski et al, Amazon Aurora—Design considerations for high throughput cloud-native relational databases, *Proc. 2017 ACM Int'l Conf. Management of Data* (SIGMOD '17), 2017, pp 1041-1052. Link
R51. Werner Vogels, Working backwards, *All Things Distributed* blog, Nov. 1, 2006. Link [Vogels06]
R52. Werner Vogels et al, Dynamo: Amazon's highly available key-value store, *Proc. 21st ACM Symp. Operating Systems Principles* (SOSP '07), *ACM SIGOPS Operating Systems Review* **41**, 6, Dec. 2007, pp 205-220. Link
R53. Kaiyuan Yang et al, Exploiting the analog properties of digital circuits for malicious hardware, *Comm. ACM* **60**, 9, Sep. 2017, pp 83-91. Link

Index

Players

Stories