

Hacker Laws

Laws, Theories, Principles, and Patterns that developers will find
useful. v0.4.0, 2025-12-29.

Dave Kerr, github.com/dwmkerr/hacker-laws

Table Of Contents

Introduction	5
Laws	5
90–9–1 Principle (1% Rule)	5
90–90 Rule	5
Amdahl’s Law	6
The Broken Windows Theory	6
Brooks’ Law	7
CAP Theorem (Brewer’s Theorem)	7
Clarke’s three laws	8
Conway’s Law	8
Cunningham’s Law	9
Dunbar’s Number	9
The Dunning-Kruger Effect	9
Fitts’ Law	10
Gall’s Law	11
Goodhart’s Law	11
Hanlon’s Razor	12
Hick’s Law (Hick-Hyman Law)	12
Hofstadter’s Law	13
Hutber’s Law	13
The Hype Cycle & Amara’s Law	13
Hyrum’s Law (The Law of Implicit Interfaces)	14
Input-Process-Output (IPO)	14
Kernighan’s Law	15
Koomey’s Law	15
Linus’s Law	16
Metcalf’s Law	16
Moore’s Law	17
Murphy’s Law / Sod’s Law	17
Occam’s Razor	18

Parkinson's Law	18
Premature Optimization Effect	18
Putt's Law	18
Reed's Law	19
The Bitter Lesson	19
The Ringelmann Effect	20
The Law of Conservation of Complexity (Tesler's Law)	20
The Law of Demeter	20
The Law of Leaky Abstractions	20
The Law of the Instrument	21
The Law of Triviality	22
The Unix Philosophy	22
The Scout Rule	22
The Spotify Model	23
The Two Pizza Rule	23
Twyman's law	23
Wadler's Law	24
Wheaton's Law	24
Principles	25
All Models Are Wrong (George Box's Law)	25
Chesterton's Fence	25
Kerckhoffs's principle	25
The Dead Sea Effect	26
The Dilbert Principle	26
The Pareto Principle (The 80/20 Rule)	27
The Shirky Principle	27
The Stochastic Parrot	28
The Peter Principle	28
The Robustness Principle (Postel's Law)	29
SOLID	29
The Single Responsibility Principle	29
The Open/Closed Principle	30
The Liskov Substitution Principle	30
The Interface Segregation Principle	31
The Dependency Inversion Principle	32
The DRY Principle	32
The KISS principle	33
YAGNI	33
The Fallacies of Distributed Computing	33
The Principle of Least Astonishment	34
Reading List	35
Online Resources	35
PDF eBook	35
Podcast	35
Contributors	35



Laws, Theories, Principles and Patterns for developers and technologists.

-
- My new book Effective Shell (Online Version) on Amazon (Print/Kindle)
 - Try hacker-laws.com
 - Check out my new project Terminal AI
 - Like this project? Consider buying me a coffee with a one-off donation
 - Listen to the podcast The Changelog - Laws for Hackers to Live By
 - Download the PDF eBook
 - See the Translations:

-
- Introduction
 - Laws
 - 90–9–1 Principle (1% Rule)
 - 90–90 Rule
 - Amdahl’s Law
 - The Broken Windows Theory
 - Brooks’ Law
 - CAP Theorem (Brewer’s Theorem)
 - Clarke’s three laws
 - Conway’s Law
 - Cunningham’s Law
 - Dunbar’s Number
 - The Dunning-Kruger Effect
 - Fitts’ Law
 - Gall’s Law
 - Goodhart’s Law
 - Hanlon’s Razor
 - Hick’s Law (Hick-Hyman Law)
 - Hofstadter’s Law
 - Hutber’s Law
 - The Hype Cycle & Amara’s Law
 - Hyrum’s Law (The Law of Implicit Interfaces)
 - Input-Process-Output (IPO)
 - Kernighan’s Law
 - Koomey’s Law
 - Linus’s Law
 - Metcalfe’s Law
 - Moore’s Law
 - Murphy’s Law / Sod’s Law
 - Occam’s Razor

- Parkinson’s Law
- Premature Optimization Effect
- Putt’s Law
- Reed’s Law
- The Bitter Lesson
- The Ringelmann Effect
- The Law of Conservation of Complexity (Tesler’s Law)
- The Law of Demeter
- The Law of Leaky Abstractions
- The Law of the Instrument
- The Law of Triviality
- The Unix Philosophy
- The Scout Rule
- The Spotify Model
- The Two Pizza Rule
- Twyman’s law
- Wadler’s Law
- Wheaton’s Law
- Principles
 - All Models Are Wrong (George Box’s Law)
 - Chesterton’s Fence
 - Kerckhoffs’s principle
 - The Dead Sea Effect
 - The Dilbert Principle
 - The Pareto Principle (The 80/20 Rule)
 - The Shirky Principle
 - The Stochastic Parrot
 - The Peter Principle
 - The Robustness Principle (Postel’s Law)
 - SOLID
 - The Single Responsibility Principle
 - The Open/Closed Principle
 - The Liskov Substitution Principle
 - The Interface Segregation Principle
 - The Dependency Inversion Principle
 - The DRY Principle
 - The KISS principle
 - YAGNI
 - The Fallacies of Distributed Computing
 - The Principle of Least Astonishment
- Reading List
- Online Resources
- PDF eBook
- Podcast

Introduction

There are lots of laws which people discuss when talking about development. This repository is a reference and overview of some of the most common ones. Please share and submit PRs!

Warning: This repo contains an explanation of some laws, principles and patterns, but does not *advocate* for any of them. Whether they should be applied will always be a matter of debate, and greatly dependent on what you are working on.

Laws

Laws can be opinions on inevitabilities in the world of software engineering, or wry observations on unavoidable realities.

90–9–1 Principle (1% Rule)

1% Rule on Wikipedia

The 90-9-1 principle suggests that within an internet community such as a wiki, 90% of participants only consume content, 9% edit or modify content and 1% of participants add content.

Real-world examples:

- A 2014 study of four digital health social networks found the top 1% created 73% of posts, the next 9% accounted for an average of ~25% and the remaining 90% accounted for an average of 2% (Reference)

See Also:

- Pareto principle

90–90 Rule

90-90 Rule on Wikipedia

The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.

A wry reinterpretation of the Pareto Principle (or 80-20 rule) that highlights the real-world challenges of completing engineering work. This sentiment is also echoed in Hofstadter's Law.

See also:

- Hofstadter's Law
- The Pareto Principle

Amdahl's Law

Amdahl's Law on Wikipedia

Amdahl's Law is a formula which shows the *potential speedup* of a computational task which can be achieved by increasing the resources of a system. Normally used in parallel computing, it can predict the actual benefit of increasing the number of processors, which is limited by the parallelisability of the program.

Best illustrated with an example. If a program is made up of two parts, part A, which must be executed by a single processor, and part B, which can be parallelised, then we see that adding multiple processors to the system executing the program can only have a limited benefit. It can potentially greatly improve the speed of part B - but the speed of part A will remain unchanged.

The diagram below shows some examples of potential improvements in speed:

As can be seen, even a program which is 50% parallelisable will benefit very little beyond 10 processing units, whereas a program which is 95% parallelisable can still achieve significant speed improvements with over a thousand processing units.

As Moore's Law slows, and the acceleration of individual processor speed slows, parallelisation is key to improving performance. Graphics programming is an excellent example - with modern Shader based computing, individual pixels or fragments can be rendered in parallel - this is why modern graphics cards often have many thousands of processing cores (GPUs or Shader Units).

See also:

- Brooks' Law
- Moore's Law

The Broken Windows Theory

The Broken Windows Theory on Wikipedia

The Broken Windows Theory suggests that visible signs of crime (or lack of care of an environment) lead to further and more serious crimes (or further deterioration of the environment).

This theory has been applied to software development, suggesting that poor quality code (or Technical Debt) can lead to a perception that efforts to improve quality may be ignored or undervalued, thus leading to further poor quality code. This effect cascades leading to a great decrease in quality over time.

See also:

- Technical Debt

Examples:

- The Pragmatic Programming: Software Entropy
- Coding Horror: The Broken Window Theory
- OpenSource: Joy of Programming - The Broken Window Theory

Brooks' Law

Brooks' Law on Wikipedia

Adding human resources to a late software development project makes it later.

This law suggests that in many cases, attempting to accelerate the delivery of a project which is already late, by adding more people, will make the delivery even later. Brooks is clear that this is an over-simplification, however, the general reasoning is that given the ramp-up time of new resources and the communication overheads, in the immediate short-term velocity decreases. Also, many tasks may not be divisible, i.e. easily distributed between more resources, meaning the potential velocity increase is also lower.

The common phrase in delivery “Nine women can’t make a baby in one month” relates to Brooks' Law, in particular, the fact that some kinds of work are not divisible or parallelisable.

This is a central theme of the book ‘The Mythical Man Month’.

See also:

- Death March
- Reading List: The Mythical Man Month

CAP Theorem (Brewer's Theorem)

The CAP Theorem (defined by Eric Brewer) states that for a distributed data store only two out of the following three guarantees (at most) can be made:

- Consistency: when reading data, every request receives the *most recent* data or an error is returned
- Availability: when reading data, every request receives a *non error response*, without the guarantee that it is the *most recent* data
- Partition Tolerance: when an arbitrary number of network requests between nodes fail, the system continues to operate as expected

The core of the reasoning is as follows. It is impossible to guarantee that a network partition will not occur (see The Fallacies of Distributed Computing). Therefore in the case of a partition we can either cancel the operation (increasing consistency and decreasing availability) or proceed (increasing availability but decreasing consistency).

The name comes from the first letters of the guarantees (Consistency, Availability, Partition Tolerance). Note that it is very important to be aware that this does *not* relate to *ACID*, which has a different definition of consistency. More recently,

PACELC theorem has been developed which adds constraints for latency and consistency when the network is *not* partitioned (i.e. when the system is operating as expected).

Most modern database platforms acknowledge this theorem implicitly by offering the user of the database the option to choose between whether they want a highly available operation (which might include a ‘dirty read’) or a highly consistent operation (for example a ‘quorum acknowledged write’).

Real world examples:

- Inside Google Cloud Spanner and the CAP Theorem - Goes into the details of how Cloud Spanner works, which appears at first to seem like a platform which has *all* of the guarantees of CAP, but under the hood is essentially a CP system.

See also:

- ACID
- The Fallacies of Distributed Computing
- PACELC

Clarke’s three laws

Clarke’s three laws on Wikipedia

Arthur C. Clarke, an british science fiction writer, formulated three adages that are known as Clarke’s three laws. The third law is the best known and most widely cited.

These so-called laws are:

- When a distinguished but elderly scientist states that something is possible, they are almost certainly right. When they state that something is impossible, they are very probably wrong.
- The only way of discovering the limits of the possible is to venture a little way past them into the impossible.
- Any sufficiently advanced technology is indistinguishable from magic.

Conway’s Law

Conway’s Law on Wikipedia

This law suggests that the technical boundaries of a system will reflect the structure of the organisation. It is commonly referred to when looking at organisation improvements, Conway’s Law suggests that if an organisation is structured into many small, disconnected units, the software it produces will be. If an organisation is built more around ‘verticals’ which are oriented around features or services, the software systems will also reflect this.

See also:

- The Spotify Model

Cunningham’s Law

Cunningham’s Law on Wikipedia

The best way to get the right answer on the Internet is not to ask a question, it’s to post the wrong answer.

According to Steven McGeady, Ward Cunningham advised him in the early 1980s: “The best way to get the right answer on the Internet is not to ask a question, it’s to post the wrong answer.” McGeady dubbed this Cunningham’s law, though Cunningham denies ownership calling it a “misquote.” Although originally referring to interactions on Usenet, the law has been used to describe how other online communities work (e.g., Wikipedia, Reddit, Twitter, Facebook).

See also:

- XKCD 386: “Duty Calls”

Dunbar’s Number

Dunbar’s Number on Wikipedia

“Dunbar’s number is a suggested cognitive limit to the number of people with whom one can maintain stable social relationships— relationships in which an individual knows who each person is and how each person relates to every other person.” There is some disagreement to the exact number. “... [Dunbar] proposed that humans can comfortably maintain only 150 stable relationships.” He put the number into a more social context, “the number of people you would not feel embarrassed about joining uninvited for a drink if you happened to bump into them in a bar.” Estimates for the number generally lay between 100 and 250.

Like stable relationships between individuals, a developer’s relationship with a codebase takes effort to maintain. When faced with large complicated projects, or ownership of many projects, we lean on convention, policy, and modeled procedure to scale. Dunbar’s number is not only important to keep in mind as an office grows, but also when setting the scope for team efforts or deciding when a system should invest in tooling to assist in modeling and automating logistical overhead. Putting the number into an engineering context, it is the number of projects (or normalized complexity of a single project) for which you would feel confident in joining an on-call rotation to support.

See also:

- Conway’s Law

The Dunning-Kruger Effect

The Dunning-Kruger Effect on Wikipedia

If you're incompetent, you can't know you're incompetent... The skills you need to produce a right answer are exactly the skills you need to recognize what a right answer is.

(David Dunning)

The Dunning–Kruger effect is a theoretical cognitive bias which was described by David Dunning and Justin Kruger in a 1999 psychological study and paper. The study suggests that people with a low level of ability at a task are likely to overestimate their ability of the task. The proposed reason for this bias is that a sufficient *awareness* of the complexity of a problem or domain is required for a person to be able to make an informed opinion of their capability to work in that domain.

The Dunning-Kruger effect has sometimes been used to describe a related, but not necessarily implied effect which could be described as “The less a person understands a domain, the more they are likely to believe they can easily solve problems in that domain, as they are more likely to see the domain as *simple*”. This more general effect is highly relevant in technology. It would suggest that people who are less familiar with a domain, such as non-technical team members or less experienced team members, are more likely to *underestimate* the effort required to solve a problem in this space.

As a person's understanding and experience in a domain grows, they may well encounter another effect, which is that they tend to *overestimate* the ability of *others* or *underestimate* their own ability, as they have become so experienced in the domain. In all cases these effects are *cognitive biases*. As with any bias, an understanding that it may be present will often be sufficient to help avoid the challenges — as when there is awareness of a bias, more inputs and opinions can be included to attempt to eliminate these biases. A closely related bias is that of Illusory superiority.

Real-world examples:

Fitts' Law

Fitts' Law on Wikipedia

Fitts' law predicts that the time required to move to a target area is a function of the distance to the target divided by the width of the target.

The consequences of this law dictate that when designing UX or UI, interactive elements should be as large as possible and the distance between the users attention area and interactive element should be as small as possible. This has consequences on design, such as grouping tasks that are commonly used with one another close.

It also formalises the concept of 'magic corners', the corners of the screen to which a user can 'sweep' their mouse to easily hit - which is where key UI elements can be placed. The Windows Start button is in a magic corner, making

it easy to select, and as an interesting contrast, the MacOS ‘close window’ button is *not* in a magic corner, making it hard to hit by mistake.

See also:

- The information capacity of the human motor system in controlling the amplitude of movement.

Gall’s Law

Gall’s Law on Wikipedia

A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system.

(John Gall)

Gall’s Law implies that attempts to *design* highly complex systems are likely to fail. Highly complex systems are rarely built in one go, but evolve instead from more simple systems.

The classic example is the world-wide-web. In its current state, it is a highly complex system. However, it was defined initially as a simple way to share content between academic institutions. It was very successful in meeting these goals and evolved to become more complex over time.

See also:

- KISS (Keep It Simple, Stupid)

Goodhart’s Law

The Goodhart’s Law on Wikipedia

Any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes.

Charles Goodhart

Also commonly referenced as:

When a measure becomes a target, it ceases to be a good measure.

Marilyn Strathern

The law states that the measure-driven optimizations could lead to devaluation of the measurement outcome itself. Overly selective set of measures (KPIs) blindly applied to a process results in distorted effect. People tend to optimize locally by “gaming” the system in order to satisfy particular metrics instead of paying attention to holistic outcome of their actions.

Real-world examples: - Assert-free tests satisfy the code coverage expectation, despite the fact that the metric intent was to create well-tested software. - Developer performance score indicated by the number of lines committed leads to unjustifiably bloated codebase.

See also: - Goodhart's Law: How Measuring The Wrong Things Drive Immoral Behaviour - Dilbert on bug-free software

Hanlon's Razor

Hanlon's Razor on Wikipedia

Never attribute to malice that which is adequately explained by stupidity.

Robert J. Hanlon

This principle suggests that actions resulting in a negative outcome were not a result of ill will. Instead the negative outcome is more likely attributed to those actions and/or the impact being not fully understood.

Hick's Law (Hick-Hyman Law)

Hick's law on Wikipedia

Decision time grows logarithmically with the number of options you can choose from.

William Edmund Hick and Ray Hyman

In the equation below, T is the time to make a decision, n is the number of options, and b is a constant which is determined by analysis of the data.

$$T = b \cdot \log_2(n + 1)$$

Figure 1: Hicks law

This law only applies when the number of options is *ordered*, for example, alphabetically. This is implied in the base two logarithm - which implies the decision maker is essentially performing a *binary search*. If the options are not well ordered, experiments show the time taken is linear.

This has significant impact in UI design; ensuring that users can easily search through options leads to faster decision making.

A correlation has also been shown in Hick's Law between IQ and reaction time as shown in Speed of Information Processing: Developmental Change and Links to Intelligence.

See also: - Fitts's Law

Hofstadter's Law

Hofstadter's Law on Wikipedia

It always takes longer than you expect, even when you take into account Hofstadter's Law.

(Douglas Hofstadter)

You might hear this law referred to when looking at estimates for how long something will take. It seems a truism in software development that we tend to not be very good at accurately estimating how long something will take to deliver.

This is from the book 'Gödel, Escher, Bach: An Eternal Golden Braid'.

See also:

- Reading List: Gödel, Escher, Bach: An Eternal Golden Braid

Hutber's Law

Hutber's Law on Wikipedia

Improvement means deterioration.

(Patrick Hutber)

This law suggests that improvements to a system will lead to deterioration in other parts, or it will hide other deterioration, leading overall to a degradation from the current state of the system.

For example, a decrease in response latency for a particular end-point could cause increased throughput and capacity issues further along in a request flow, affecting an entirely different sub-system.

The Hype Cycle & Amara's Law

The Hype Cycle on Wikipedia

We tend to overestimate the effect of a technology in the short run and underestimate the effect in the long run.

(Roy Amara)

The Hype Cycle is a visual representation of the excitement and development of technology over time, originally produced by Gartner. It is best shown with a visual:

In short, this cycle suggests that there is typically a burst of excitement around new technology and its potential impact. Teams often jump into these technologies quickly, and sometimes find themselves disappointed with the results. This might be because the technology is not yet mature enough, or real-world applications are not yet fully realised. After a certain amount of time, the capabilities

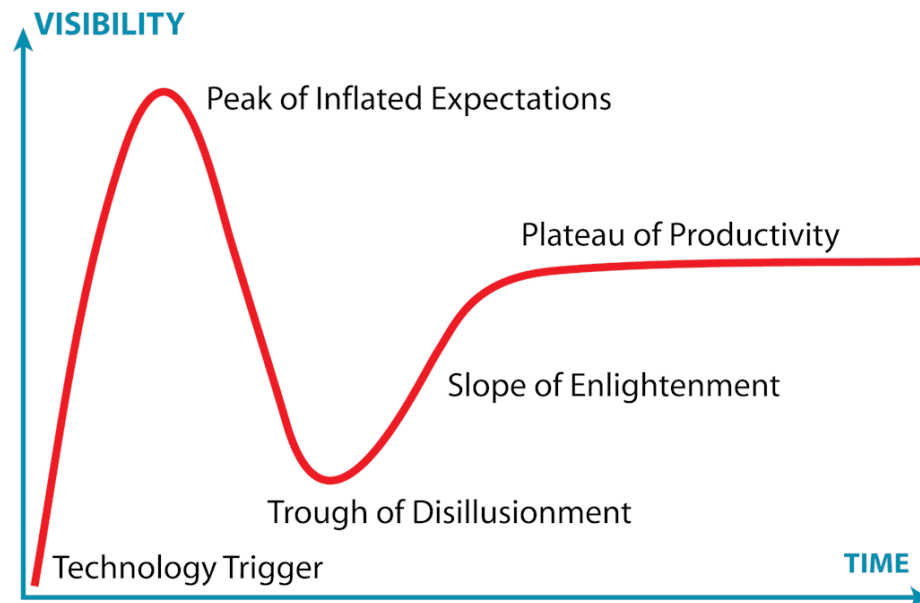


Figure 2: The Hype Cycle

of the technology increase and practical opportunities to use it increase, and teams can finally become productive. Roy Amara's quote sums this up most succinctly - "We tend to overestimate the effect of a technology in the short run and underestimate in the long run".

Hyrum's Law (The Law of Implicit Interfaces)

Hyrum's Law Online

With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody.

(Hyrum Wright)

See also:

- The Law of Leaky Abstractions
- XKCD 1172

Input-Process-Output (IPO)

Input-Process-Output on Wikipedia

Systems can be incredibly complex, but can typically be broken down into smaller parts that follow a simple pattern:

1. Input is provided
2. Some kind of processing or transformation is performed
3. Output is returned

A sort function in a programming language or system could be a classic example of the IPO pattern; where arbitrary input is sorted based on a predicate and returned back. A web server could be modelled as an IPO system, where HTTP requests are transformed into HTTP responses. A highly complex Generative AI system could likewise be modelled in this way, with user input being passed through a complex model and a response being generated.

The IPO pattern is present in different forms across almost all technological domains, from functional programming languages that explicitly follow IPO patterns to The Unix Philosophy, which suggests that highly complex systems can be built by chaining together many simple IPO programs.

See also:

- The Unix Philosophy

Kernighan's Law

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

(Brian Kernighan)

Kernighan's Law is named for Brian Kernighan and derived from a quote from Kernighan and Plauger's book *The Elements of Programming Style*:

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

While hyperbolic, Kernighan's Law makes the argument that simple code is to be preferred over complex code, because debugging any issues that arise in complex code may be costly or even infeasible.

See also:

- The KISS Principle
- The Unix Philosophy
- Occam's Razor

Koomey's Law

Koomey's Law on Wikipedia

...at a fixed computing load, the amount of battery you need will fall by a factor of two every year and a half.

(Jonathan Koomey)

In 2010 Professor Jonathan Koomey discovered that the trend in number of computations per joule of energy dissipated had been remarkably stable. This trend became known as Koomey's Law - that the amount of battery needed for a given computing load would half each 2.5 years.

Koomey performed a follow-up analysis in 2010 and found that this trend had slowed, similar to how Moore's Law had slowed. This seemed to be related to limitations around how small transistors can be made, as well as Dennard Scaling.

See also:

- Moore's Law
- Dennard Scaling

Linus's Law

Linus's Law on Wikipedia

Given enough eyeballs, all bugs are shallow.

Eric S. Raymond

This law simply states that the more people who can see a problem, the higher the likelihood that someone will have seen and solved the problem before, or something very similar.

Although it was originally used to describe the value of open-source models for projects it can be accepted for any kind of software project. It can also be extended to processes - more code reviews, more static analysis and multi-disciplined test processes will make the problems more visible and easy to identify.

A more formal statement can be:

Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and can be solved by someone who has encountered a similar problem before.

This law was named in honour of Linus Torvalds in Eric S. Raymond's book "The Cathedral and the Bazaar".

Metcalf's Law

Metcalf's Law on Wikipedia

In network theory, the value of a system grows as approximately the square of the number of users of the system.

This law is based on the number of possible pairwise connections within a system and is closely related to Reed's Law. Odlyzko and others have argued that

both Reed's Law and Metcalfe's Law overstate the value of the system by not accounting for the limits of human cognition on network effects; see Dunbar's Number.

See also: - Reed's Law - Dunbar's Number

Moore's Law

Moore's Law on Wikipedia

The number of transistors in an integrated circuit doubles approximately every two years.

Often used to illustrate the sheer speed at which semiconductor and chip technology has improved, Moore's prediction has proven to be highly accurate over from the 1970s to the late 2000s. In more recent years, the trend has changed slightly, partly due to physical limitations on the degree to which components can be miniaturised. However, advancements in parallelisation, and potentially revolutionary changes in semiconductor technology and quantum computing may mean that Moore's Law could continue to hold true for decades to come.

See also:

- Koomey's Law

Murphy's Law / Sod's Law

Murphy's Law on Wikipedia

Anything that can go wrong will go wrong.

Related to Edward A. Murphy, Jr *Murphy's Law* states that if a thing can go wrong, it will go wrong.

This is a common adage among developers. Sometimes the unexpected happens when developing, testing or even in production. This can also be related to the (more common in British English) *Sod's Law*:

If something can go wrong, it will, at the worst possible time.

These 'laws' are generally used in a comic sense. However, phenomena such as *Confirmation Bias* and *Selection Bias* can lead people to perhaps over-emphasise these laws (the majority of times when things work, they go unnoticed, failures however are more noticeable and draw more discussion).

See Also:

- Confirmation Bias
- Selection Bias

Occam's Razor

Occam's Razor on Wikipedia

Entities should not be multiplied without necessity.

William of Ockham

Occam's razor says that among several possible solutions, the most likely solution is the one with the least number of concepts and assumptions. This solution is the simplest and solves only the given problem, without introducing accidental complexity and possible negative consequences.

See also:

- YAGNI
- No Silver Bullet: Accidental Complexity and Essential Complexity

Example:

- Lean Software Development: Eliminate Waste

Parkinson's Law

Parkinson's Law on Wikipedia

Work expands so as to fill the time available for its completion.

In its original context, this Law was based on studies of bureaucracies. It may be pessimistically applied to software development initiatives, the theory being that teams will be inefficient until deadlines near, then rush to complete work by the deadline, thus making the actual deadline somewhat arbitrary.

See also:

- Hofstadter's Law

Premature Optimization Effect

Premature Optimization on WikiWeb

Premature optimization is the root of all evil.

(Donald Knuth)

However, *Premature Optimization* can be defined (in less loaded terms) as optimizing before we know that we need to.

Putt's Law

Putt's Law on Wikipedia

Technology is dominated by two types of people, those who understand what they do not manage and those who manage what they do not understand.

Putt's Law is often followed by Putt's Corollary:

Every technical hierarchy, in time, develops a competence inversion.

These statements suggest that due to various selection criteria and trends in how groups organise, there will be a number of skilled people at working levels of a technical organisations, and a number of people in managerial roles who are not aware of the complexities and challenges of the work they are managing. This can be due to phenomena such as The Peter Principle or The Dilbert Principle.

However, it should be stressed that Laws such as this are vast generalisations and may apply to *some* types of organisations, and not apply to others.

See also:

- The Peter Principle
- The Dilbert Principle

Reed's Law

Reed's Law on Wikipedia

The utility of large networks, particularly social networks, scales exponentially with the size of the network.

This law is based on graph theory, where the utility scales as the number of possible sub-groups, which is faster than the number of participants or the number of possible pairwise connections. Odlyzko and others have argued that Reed's Law overstates the utility of the system by not accounting for the limits of human cognition on network effects; see Dunbar's Number.

See also: - Metcalfe's Law - Dunbar's Number

The Bitter Lesson

The Bitter Lesson by Richard S. Sutton

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin.

Richard S. Sutton (2019)

The "Bitter Lesson", stated by Rich S. Sutton, says that scale (in terms of both data and computational power) has driven the most significant advancements in AI research, rather than the intricacies of the research methods themselves.

He goes on to suggest that this indicates we should stop trying to build simplified (or even complex) models of the mind as history has shown that these have always in the long term been failures compared to (as an example) scaling the capacity of neural networks and applying existing methods such as convolution.

The Ringelmann Effect

The Ringelmann effect on Wikipedia

The Ringelmann Effect is the tendency of an individual to become increasingly inefficient as more and more people are involved in a task. In other words, as more individuals are added to a team, the more the average individual performance decreases. Multiple causes are believed to be at work, including loss of motivation (“social loafing”) and challenges related to coordination.

See also: - Brooks’ Law

The Law of Conservation of Complexity (Tesler’s Law)

The Law of Conservation of Complexity on Wikipedia

This law states that there is a certain amount of complexity in a system which cannot be reduced.

Some complexity in a system is ‘inadvertent’. It is a consequence of poor structure, mistakes, or just bad modeling of a problem to solve. Inadvertent complexity can be reduced (or eliminated). However, some complexity is ‘intrinsic’ as a consequence of the complexity inherent in the problem being solved. This complexity can be moved, but not eliminated.

One interesting element to this law is the suggestion that even by simplifying the entire system, the intrinsic complexity is not reduced, it is *moved to the user*, who must behave in a more complex way.

The Law of Demeter

The Law of Demeter on Wikipedia

Don’t talk to strangers.

The Law of Demeter, also known as “The Principle of Least Knowledge” is a principle for software design, particularly relevant in object orientated languages.

It states that a unit of software should talk only to its immediate collaborators. An object **A** with a reference to object **B** can call its methods, but if **B** has a reference to object **C**, **A** should not call **C**s methods. So, if **C** has a `doThing()` method, **A** should not invoke it directly; `B.getC().doThis()`.

Following this principal limits the scope of changes, making them easier and safer in future.

The Law of Leaky Abstractions

The Law of Leaky Abstractions on Joel on Software

All non-trivial abstractions, to some degree, are leaky.

(Joel Spolsky)

This law states that abstractions, which are generally used in computing to simplify working with complicated systems, will in certain situations ‘leak’ elements of the underlying system, this making the abstraction behave in an unexpected way.

The example above can become more complex when *more* abstractions are introduced. The Linux operating system allows files to be accessed over a network but represented locally as ‘normal’ files. This abstraction will ‘leak’ if there are network failures. If a developer treats these files as ‘normal’ files, without considering the fact that they may be subject to network latency and failures, the solutions will be buggy.

The article describing the law suggests that an over-reliance on abstractions, combined with a poor understanding of the underlying processes, actually makes dealing with the problem at hand *more* complex in some cases.

See also:

- Hyrum’s Law

Real-world examples:

- Photoshop Slow Startup - an issue I encountered in the past. Photoshop would be slow to startup, sometimes taking minutes. It seems the issue was that on startup it reads some information about the current default printer. However, if that printer is actually a network printer, this could take an extremely long time. The *abstraction* of a network printer being presented to the system similar to a local printer caused an issue for users in poor connectivity situations.

The Law of the Instrument

The Law of the Instrument

I call it the law of the instrument, and it may be formulated as follows: Give a small boy a hammer, and he will find that everything he encounters needs pounding.

Abraham Kaplan

If all you have is a hammer, everything looks like a nail.

Abraham Maslow

In the context of computer programming, this law suggests that people tend to use tools that are familiar with, rather than the best possible tool. This over-reliance on a familiar tool is an anti-pattern referred to as ‘the golden hammer’.

See also:

- Avoiding the law of the instrument

- Anti-Pattern - The Golden Hammer

The Law of Triviality

The Law of Triviality on Wikipedia

This law suggests that groups will give far more time and attention to trivial or cosmetic issues rather than serious and substantial ones.

The common fictional example used is that of a committee approving plans for nuclear power plant, who spend the majority of their time discussing the structure of the bike shed, rather than the far more important design for the power plant itself. It can be difficult to give valuable input on discussions about very large, complex topics without a high degree of subject matter expertise or preparation. However, people want to be seen to be contributing valuable input. Hence a tendency to focus too much time on small details, which can be reasoned about easily, but are not necessarily of particular importance.

The fictional example above led to the usage of the term ‘Bike Shedding’ as an expression for wasting time on trivial details. A related term is ‘Yak Shaving,’ which connotes a seemingly irrelevant activity that is part of a long chain of prerequisites to the main task.

The Unix Philosophy

The Unix Philosophy on Wikipedia

The Unix Philosophy is that software components should be small, and focused on doing one specific thing well. This can make it easier to build systems by composing together small, simple, well-defined units, rather than using large, complex, multi-purpose programs.

Modern practices like ‘Microservice Architecture’ can be thought of as an application of this law, where services are small, focused and do one specific thing, allowing complex behaviour to be composed of simple building blocks.

The Scout Rule

The Scout Rule on O’Reilly

Always leave the code better than you found it.

(Robert C. Martin (Uncle Bob))

Based on the “Scout Rule”, which is “always leave the campground cleaner than you found it”, the Scout Rule in programming is simply “always leave the code cleaner than you found it”.

This was introduced in the first chapter of the book Clean Code by Bob Martin. The rule suggests that developers should perform ‘optimistic refactoring’, which means to endeavour to improve the overall quality of the code when you work

on it. If you see a mistake, attempt to fix it or clean it up. However, when making changes to code which seems incorrect, it may be worth remembering Chesterton's Fence!

See also:

- Reading List: Clean Code
- Chesterton's Fence
- The Broken Windows Theory

<https://www.amazon.sg/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882>

The Spotify Model

The Spotify Model on Spotify Labs

The Spotify Model is an approach to team and organisation structure which has been popularised by 'Spotify'. In this model, teams are organised around features, rather than technologies.

The Spotify Model also popularises the concepts of Tribes, Guilds, Chapters, which are other components of their organisation structure.

Members of the organisation have described that the actual meaning of these groups changes, evolves and is an on-going experiment. The fact that the model is a *process in motion*, rather than a fixed model continues to lead to varying interpretations of the structure, which may be based on presentations given by employees at conferences. This means 'snapshots' may be 're-packaged' by third parties as a *fixed structure*, with the fact that the model is dynamic being lost.

The Two Pizza Rule

If you can't feed a team with two pizzas, it's too large.

(Jeff Bezos)

This rule suggests that regardless of the size of the company, teams should be small enough to be fed by two pizzas. Attributed to Jeff Bezos and Amazon, this belief suggests that large teams are inherently inefficient. This is supported by the fact that as the team size increases linearly, the links between people increases quadratically; thus the cost of coordinating and communicating also grows quadratically. If this cost of coordination is essentially overhead, then smaller teams should be preferred.

The number of links between people can be expressed as $n(n-1)/2$ where n = number of people.

Twyman's law

Twyman's Law on Wikipedia

The more unusual or interesting the data, the more likely they are to have been the result of an error of one kind or another.

This law suggests that when there are particularly unusual data points, it is more likely that they are the result of errors or manipulation. For example, if a dataset of long-jump results from a sporting event showed a maximum value of 20 meters (more than twice the world record), it is more likely to be due to an error (such as recording a value in feet rather than meters) than due to an unusually long jump. It is also more likely in this case that the results could have been manipulated.

See also:

- Sagan Standard

Wadler's Law

Wadler's Law on wiki.haskell.org

In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.

0. Semantics
1. Syntax
2. Lexical syntax
3. Lexical syntax of comments

(In short, for every hour spent on semantics, 8 hours will be spent on the syntax of comments).

Similar to The Law of Triviality, Wadler's Law states what when designing a language, the amount of time spent on language structures is disproportionately high in comparison to the importance of those features.

See also:

- The Law of Triviality

Wheaton's Law

The Link

The Official Day

Don't be a dick.

Wil Wheaton

Coined by Wil Wheaton (Star Trek: The Next Generation, The Big Bang Theory), this simple, concise, and powerful law aims for an increase in harmony and respect within a professional organization. It can be applied when speaking with coworkers, performing code reviews, countering other points of view, critiquing, and in general, most professional interactions humans have with each other.

Principles

Principles are generally more likely to be guidelines relating to design.

All Models Are Wrong (George Box's Law)

All Models Are Wrong

All models are wrong, but some are useful.

George Box

This principle suggests that all models of systems are flawed, but that as long as they are not *too* flawed they may be useful. This principle has its roots in statistics but applies to scientific and computing models as well.

A fundamental requirement of most software is to model a system of some kind. Regardless of whether the system being modeled is a computer network, a library, a graph of social connections or any other kind of system, the designer will have to decide an appropriate level of detail to model. Excessive detail may lead to too much complexity, too little detail may prevent the model from being functional.

See also:

- The Law of Leaky Abstractions

Chesterton's Fence

Chesterton's Fence on Wikipedia

Reforms should not be made until the reasoning behind the existing state of affairs is understood.

This principle is relevant in software engineering when removing technical debt. Each line of a program was originally written by someone for some reason. Chesterton's Fence suggests that one should try to understand the context and meaning of the code fully, before changing or removing it, even if at first glance it seems redundant or incorrect.

The name of this principle comes from a story by G.K. Chesterton. A man comes across a fence crossing the middle of the road. He complains to the mayor that this useless fence is getting in the way, and asks to remove it. The mayor asks why the fence is there in the first place. When the man says he doesn't know, the mayor says, "If you don't know its purpose, I certainly won't let you remove it. Go and find out the use of it, and then I may let you destroy it."

Kerckhoffs's principle

Kerckhoffs's principle on Wikipedia

“...design your system assuming that your opponents know it in detail.”

Steven M. Bellovin's formulation of Kerckhoff's Principle

This principle of cryptography was an axiom created by cryptographer Auguste Kerckhoffs. He stated that a cryptosystem should be secure, even if everything about the system, except the key, is public knowledge. Not to be confused with “*security through obscurity*”.

The gold standard for any secret-keeping system is that implementation details should be publicly distributed, without sacrificing or compromising security of said system.

The history of cryptography has shown that open discussion and analysis of cryptographic systems leads to better and more secure systems - as researchers are able to test for and expose potential vulnerabilities.

- Shannon's Maxim

The Dead Sea Effect

The Dead Sea Effect on Bruce F. Webster

“... [T]he more talented and effective IT engineers are the ones most likely to leave - to evaporate ... [those who tend to] remain behind [are] the ‘residue’ — the least talented and effective IT engineers.”

Bruce F. Webster

The “Dead Sea Effect” suggests that in any organisation, the skills/talent/efficacy of engineers is often inversely proportional to their time in the company.

Typically, highly skilled engineers find it easy to gain employment elsewhere and are the first to do so. Engineers who have obsolete or weak skills will tend to remain with the company, as finding employment elsewhere is difficult. This is particularly pronounced if they have gained incremental pay rises over their time in the company, as it can be challenging to get equivalent remuneration elsewhere.

The Dilbert Principle

The Dilbert Principle on Wikipedia

Companies tend to systematically promote incompetent employees to management to get them out of the workflow.

Scott Adams

A management concept developed by Scott Adams (creator of the Dilbert comic strip), the Dilbert Principle is inspired by The Peter Principle. Under the Dilbert Principle, employees who were never competent are promoted to management in

order to limit the damage they can do. Adams first explained the principle in a 1995 Wall Street Journal article, and expanded upon it in his 1996 business book, *The Dilbert Principle*.

See Also:

- The Peter Principle
- Putt's Law

The Pareto Principle (The 80/20 Rule)

The Pareto Principle on Wikipedia

Most things in life are not distributed evenly.

The Pareto Principle suggests that in some cases, the majority of results come from a minority of inputs:

- 80% of a certain piece of software can be written in 20% of the total allocated time (conversely, the hardest 20% of the code takes 80% of the time)
- 20% of the effort produces 80% of the result
- 20% of the work creates 80% of the revenue
- 20% of the bugs cause 80% of the crashes
- 20% of the features cause 80% of the usage

In the 1940s American-Romanian engineer Dr. Joseph Juran, who is widely credited with being the father of quality control, began to apply the Pareto principle to quality issues.

This principle is also known as: The 80/20 Rule, The Law of the Vital Few, and The Principle of Factor Sparsity.

Real-world examples:

- In 2002 Microsoft reported that by fixing the top 20% of the most-reported bugs, 80% of the related errors and crashes in windows and office would become eliminated (Reference).

The Shirky Principle

The Shirky Principle explained

Institutions will try to preserve the problem to which they are the solution.

Clay Shirky

The Shirky Principle suggests that complex solutions - a company, an industry, or a technology - can become so focused on the problem that they are solving, that they can inadvertently perpetuate the problem itself. This may be deliberate (a company striving to find new nuances to a problem which justify continued

development of a solution), or inadvertent (being unable or unwilling to accept or build a solution which solves the problem completely or obviates it).

Related to:

- Upton Sinclair’s famous line, “*It is difficult to get a man to understand something, when his salary depends upon his not understanding it!*”
- Clay Christensen’s *The Innovator’s Dilemma*

See also:

- Pareto Principle

The Stochastic Parrot

On the Dangers of Stochastic Parrots - Bender, Gebru, et al. (2021)

Contrary to how it may seem when we observe its output, an LM is a system for haphazardly stitching together sequences of linguistic forms it has observed in its vast training data, according to probabilistic information about how they combine, but without any reference to meaning: a stochastic parrot.

Emily M. Bender, Timnit Gebru, et al. (2021)

The term argues that Large Language Models (LLMs) produce statistically likely sequences of text based on training data, without genuine comprehension. Essentially - confident-sounding output is not evidence of correctness or understanding.

Models can (and do) “hallucinate” - producing plausible sounding output or confidently making statements which are demonstrably wrong. This does not devalue these models, but highlights important characteristics which must be accounted for when using them.

See also:

- The Bitter Lesson
- All Models Are Wrong (George Box’s Law)

The Peter Principle

The Peter Principle on Wikipedia

People in a hierarchy tend to rise to their “level of incompetence”.

Laurence J. Peter

A management concept developed by Laurence J. Peter, the Peter Principle observes that people who are good at their jobs are promoted, until they reach a level where they are no longer successful (their “level of incompetence”). At this point, as they are more senior, they are less likely to be removed from the organisation (unless they perform spectacularly badly) and will continue to

reside in a role which they have few intrinsic skills at, as their original skills which made them successful are not necessarily the skills required for their new jobs.

This is of particular interest to engineers - who initially start out in deeply technical roles, but often have a career path which leads to *managing* other engineers - which requires a fundamentally different skill set.

See Also:

- The Dilbert Principle
- Putt's Law

The Robustness Principle (Postel's Law)

The Robustness Principle on Wikipedia

Be conservative in what you do, be liberal in what you accept from others.

Often applied in server application development, this principle states that what you send to others should be as minimal and conformant as possible, but you should aim to allow non-conformant input if it can be processed.

The goal of this principle is to build systems which are robust, as they can handle poorly formed input if the intent can still be understood. However, there are potentially security implications of accepting malformed input, particularly if the processing of such input is not well tested. These implications and other issues are described by Eric Allman in The Robustness Principle Reconsidered.

Allowing non-conformant input, in time, may undermine the ability of protocols to evolve as implementors will eventually rely on this liberality to build their features.

See Also:

- Hyrum's Law

SOLID

This is an acronym, which refers to:

These are key principles in Object-Oriented Programming. Design principles such as these should be able to aid developers build more maintainable systems.

The Single Responsibility Principle

The Single Responsibility Principle on Wikipedia

Every module or class should have a single responsibility only.

The first of the ‘SOLID’ principles. This principle suggests that modules or classes should do one thing and one thing only. In more practical terms, this means that a single, small change to a feature of a program should require a change in one component only. For example, changing how a password is validated for complexity should require a change in only one part of the program.

Theoretically, this should make the code more robust, and easier to change. Knowing that a component being changed has a single responsibility only means that *testing* that change should be easier. Using the earlier example, changing the password complexity component should only be able to affect the features which relate to password complexity. It can be much more difficult to reason about the impact of a change to a component which has many responsibilities.

See also:

- Object-Oriented Programming
- SOLID

The Open/Closed Principle

The Open/Closed Principle on Wikipedia

Entities should be open for extension and closed for modification.

The second of the ‘SOLID’ principles. This principle states that entities (which could be classes, modules, functions and so on) should be able to have their behaviour *extended*, but that their *existing* behaviour should not be able to be modified.

As a hypothetical example, imagine a module which is able to turn a Markdown document into HTML. Now imagine there is a new syntax added to the Markdown specification, which adds support for mathematical equations. The module should be *open to extension* to implement the new mathematics syntax. However, existing syntax implementations (like paragraphs, bullets, etc) should be *closed for modification*. They already work, we don’t want people to change them.

This principle has particular relevance for object-oriented programming, where we may design objects to be easily extended, but would avoid designing objects which can have their existing behaviour changed in unexpected ways.

See also:

- Object-Oriented Programming
- SOLID

The Liskov Substitution Principle

The Liskov Substitution Principle on Wikipedia

It should be possible to replace a type with a subtype, without breaking the system.

The third of the ‘SOLID’ principles. This principle states that if a component relies on a type, then it should be able to use subtypes of that type, without the system failing or having to know the details of what that subtype is.

As an example, imagine we have a method which reads an XML document from a structure which represents a file. If the method uses a base type ‘file’, then anything which derives from ‘file’ should be usable in the function. If ‘file’ supports seeking in reverse, and the XML parser uses that function, but the derived type ‘network file’ fails when reverse seeking is attempted, then the ‘network file’ would be violating the principle.

This principle has particular relevance for object-oriented programming, where type hierarchies must be modeled carefully to avoid confusing users of a system.

See also:

- Object-Oriented Programming
- SOLID

The Interface Segregation Principle

The Interface Segregation Principle on Wikipedia

No client should be forced to depend on methods it does not use.

The fourth of the ‘SOLID’ principles. This principle states that consumers of a component should not depend on functions of that component which it doesn’t actually use.

As an example, imagine we have a method which reads an XML document from a structure which represents a file. It only needs to read bytes, move forwards or move backwards in the file. If this method needs to be updated because an unrelated feature of the file structure changes (such as an update to the permissions model used to represent file security), then the principle has been invalidated. It would be better for the file to implement a ‘seekable-stream’ interface, and for the XML reader to use that.

This principle has particular relevance for object-oriented programming, where interfaces, hierarchies and abstract types are used to minimise the coupling between different components. Duck typing is a methodology which enforces this principle by eliminating explicit interfaces.

See also:

- Object-Oriented Programming
- SOLID
- Duck Typing
- Decoupling

The Dependency Inversion Principle

The Dependency Inversion Principle on Wikipedia

High-level modules should not be dependent on low-level implementations.

The fifth of the ‘SOLID’ principles. This principle states that higher-level orchestrating components should not have to know the details of their dependencies.

As an example, imagine we have a program which read metadata from a website. We would assume that the main component would have to know about a component to download the webpage content, then a component which can read the metadata. If we were to take dependency inversion into account, the main component would depend only on an abstract component which can fetch byte data, and then an abstract component which would be able to read metadata from a byte stream. The main component would not know about TCP/IP, HTTP, HTML, etc.

This principle is complex, as it can seem to ‘invert’ the expected dependencies of a system (hence the name). In practice, it also means that a separate orchestrating component must ensure the correct implementations of abstract types are used (e.g. in the previous example, *something* must still provide the metadata reader component a HTTP file downloader and HTML meta tag reader). This then touches on patterns such as Inversion of Control and Dependency Injection.

See also:

- Object-Oriented Programming
- SOLID
- Inversion of Control
- Dependency Injection

The DRY Principle

The DRY Principle on Wikipedia

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

DRY is an acronym for *Don't Repeat Yourself*. This principle aims to help developers reducing the repetition of code and keep the information in a single place and was cited in 1999 by Andrew Hunt and Dave Thomas in the book *The Pragmatic Programmer*

The opposite of DRY would be *WET* (Write Everything Twice or We Enjoy Typing).

In practice, if you have the same piece of information in two (or more) different places, you can use DRY to merge them into a single one and reuse it wherever you want/need.

See also:

- The Pragmatic Programmer

The KISS principle

KISS on Wikipedia

Keep it simple, stupid

The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided. Originating in the U.S. Navy in 1960, the phrase has been associated with aircraft engineer Kelly Johnson.

The principle is best exemplified by the story of Johnson handing a team of design engineers a handful of tools, with the challenge that the jet aircraft they were designing must be repairable by an average mechanic in the field under combat conditions with only these tools. Hence, the “stupid” refers to the relationship between the way things break and the sophistication of the tools available to repair them, not the capabilities of the engineers themselves.

See also:

- Gall’s Law

YAGNI

YAGNI on Wikipedia

Always implement things when you actually need them, never when you just foresee that you need them.

(Ron Jeffries) (XP co-founder and author of the book “Extreme Programming Installed”)

This *Extreme Programming* (XP) principle suggests developers should only implement functionality that is needed for the immediate requirements, and avoid attempts to predict the future by implementing functionality that might be needed later.

Adhering to this principle should reduce the amount of unused code in the codebase, and avoid time and effort being wasted on functionality that brings no value.

See also:

- Reading List: Extreme Programming Installed

The Fallacies of Distributed Computing

The Fallacies of Distributed Computing on Wikipedia

Also known as *Fallacies of Networked Computing*, the Fallacies are a list of conjectures (or beliefs) about distributed computing, which can lead to failures in software development. The assumptions are:

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

The first four items were listed by Bill Joy and Tom Lyon around 1991 and first classified by James Gosling as the “Fallacies of Networked Computing”. L. Peter Deutsch added the 5th, 6th and 7th fallacies. In the late 90's Gosling added the 8th fallacy.

The group was inspired by what was happening at the time inside Sun Microsystems.

These fallacies should be considered carefully when designing code which is resilient; assuming any of these fallacies can lead to flawed logic which fails to deal with the realities and complexities of distributed systems.

See also:

- Foraging for the Fallacies of Distributed Computing (Part 1) - Vaidehi Joshi on Medium

The Principle of Least Astonishment

The Principle of Least Astonishment on Wikipedia

People are part of the system. The design should match the user's experience, expectations, and mental models.

Frans Kaashoek

This principle proposes that systems and interfaces should be designed in a way that features and functionality is easily discovered and matches users expectations. Features that ‘surprise’ users should be discouraged in favour of features that can be intuitively reasoned about based on existing patterns and practices.

Many examples are present in user interfaces, such as a ‘pull down’ gesture on a mobile application to refresh content. Another example would be command line tools, where many standards exist for how parameters are named, common parameters that should be available and so on.

See also:

- Convention Over Configuration

Reading List

If you have found these concepts interesting, you may enjoy the following books.

- Clean Code - Robert C. Martin - One of the most well respected books on how to write clean, maintainable code.
- Extreme Programming Installed - Ron Jeffries, Ann Anderson, Chet Hendrikson - Covers the core principles of Extreme Programming.
- Gödel, Escher, Bach: An Eternal Golden Braid - Douglas R. Hofstadter. - This book is difficult to classify. Hofstadter's Law is from the book.
- Structure and Interpretation of Computer Programs - Harold Abelson, Gerald Jay Sussman, Julie Sussman - If you were a comp sci or electrical engineering student at MIT or Cambridge this was your intro to programming. Widely reported as being a turning point in people's lives.
- The Cathedral and the Bazaar - Eric S. Raymond - a collection of essays on open source. This book was the source of Linus's Law.
- The Dilbert Principle - Scott Adams - A comic look at corporate America, from the author who created the Dilbert Principle.
- The Mythical Man Month - Frederick P. Brooks Jr. - A classic volume on software engineering. Brooks' Law is a central theme of the book.
- The Peter Principle - Lawrence J. Peter - Another comic look at the challenges of larger organisations and people management, the source of The Peter Principle.

Online Resources

Some useful resources and reading.

- CB Insights: 8 Laws Driving Success In Tech: Amazon's 2-Pizza Rule, The 80/20 Principle, & More - an interesting write up of some laws which have been highly influential in technology.

PDF eBook

The project is available as a PDF eBook, download the latest PDF eBook with this link or check the release page for older versions.

A new version of the eBook is created automatically when a new version tag is pushed.

Podcast

Hacker Laws has been featured in The Changelog, you can check out the Podcast episode with the link below:

Contributors

Thanks goes to these wonderful people (emoji key):

Amir Hemmati

This project follows the all-contributors specification. Contributions of any kind welcome!