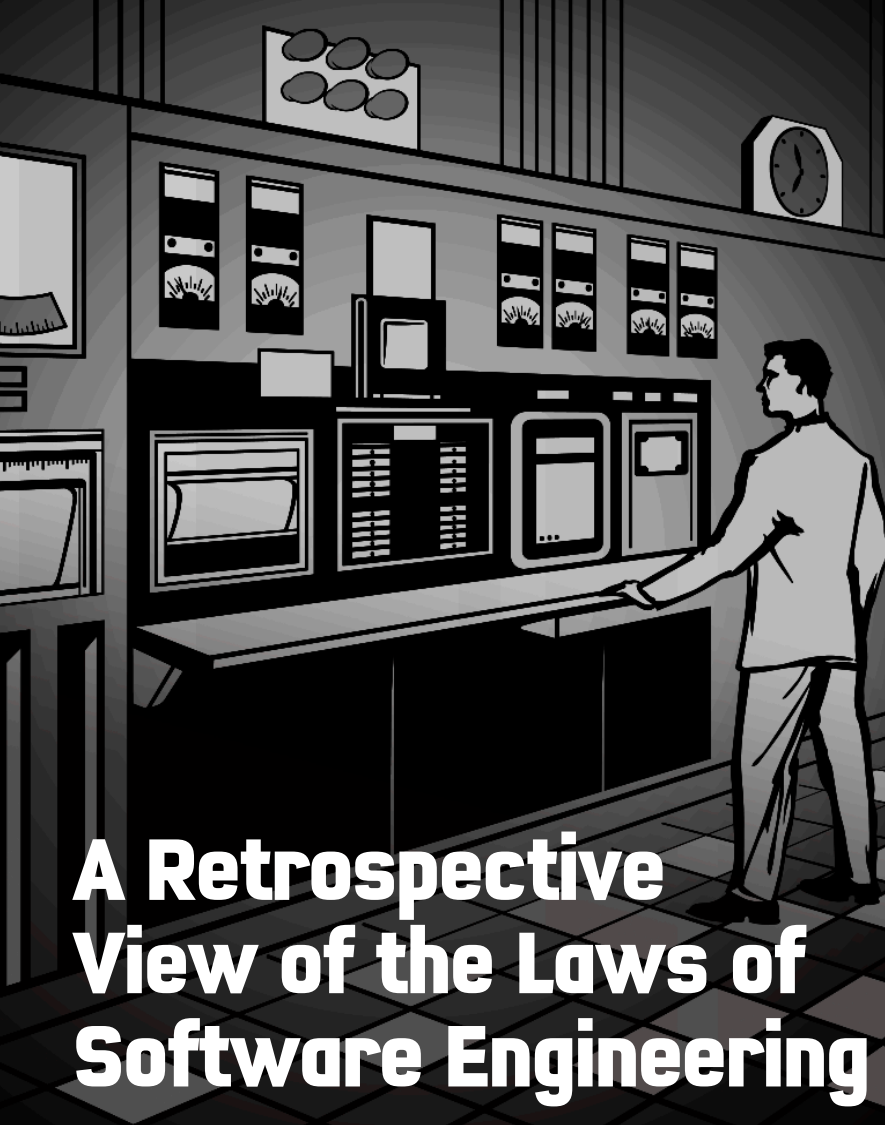


A Retrospective View of the Laws of Software Engineering

Capers Jones, VP and CTO, Namcook Analytics LLC

Abstract. Software development is now more than 60 years of age. A number of interesting laws and observations have been created by software engineering researchers and by some academics. This short paper summarizes these laws and makes observations about the data and facts that underlie them. The laws discussed in this paper are in alphabetical order.

Many of these laws did not originate with software but are taken from physics and other scientific fields. However, they are included because they seem relevant to software development.

Introduction

In 2017, software engineering is still based on custom designs and manual coding. That puts software on about the same level of manufacturing sophistication as firearms in 1784, before Eli Whitney introduced standard reusable parts and changed manufacturing forever.

It is obvious that custom designs and manual coding are intrinsically expensive and error prone, no matter what methodologies or programming languages are used.

This short paper attempts to consolidate the known factors of software engineering circa 2016. The factors are in alphabetical order. Because of the labor-intensive manual methods used to build software, many of the laws are related to problems and software failures.

Some of the laws did not originate in software but are much older and are derived from physics, chemistry, and other disciplines.

Bernoulli's Principle

- Velocity is greatest where density is least.

This is actually a law of fluid dynamics that refers to the flow of viscous liquids. However, it also applies to traffic patterns and has been used to optimize traffic flow through tunnels. It seems to apply to software as well because the work of smaller teams proceeds faster than the work of larger teams. This tends to add credence to the Agile concept of small teams.

Boehm's First Law

- Errors are more frequent during requirements and design activities and are more expensive the later they are removed.

Requirements and design errors do outnumber code errors. However, cost per defect stays flat from testing through maintenance. The cost per defect metric penalizes quality and achieves lowest values for the buggiest software. For zero defect software, the cost per defect is infinity since testing is still necessary. Defect removal cost per function point is the best choice for quality economic analysis. The reason cost per defect seems to rise is because of fixed costs. If it costs $10,000 to write and run 100 test cases and 50 bugs are fixed for another $10,000, the cost per defect is $200. If it costs $10,000 to write and run 100 test cases and only 1 bug is fixed for another $200, the cost per defect is $10,200. Writing and running test cases are fixed costs.

Boehm's Second Law

- Prototyping significantly reduces requirements and design errors, especially for user errors.

Empirical data supports this law. However, inspections and static analysis also reduce defects. A caveat is that prototypes are about 10 percent of the size of the planned system. For an application of 1,000 function points, the prototype would be about 100 function points and easily built. For a massive application of 100,000 function points, the prototype itself would be a large system of 10,000 function points. This leads to the conclusion that large systems are best done using incremental development if possible.

Brooks' Law

- Adding people to a late software project makes it later.

Empirical data supports this law to a certain degree. The complexity of communication channels increases with application size and team size. The larger the application, the more difficult it is to recover from schedule delays. For small projects with fewer than five team members, adding one more experienced person will not stretch the schedule, but adding a novice will. Projects that build large applications with more than 100 team members almost always run late due to poor quality control and poor change control. Adding people tends to slow things down due to complex communication channels and delays for training.

Buddha's Third Law

- All objects composed of component parts are fated to decay.

The historical Buddha, Sakyamuni, was born in Northern India in 525 B.C. He, of course, founded a major religion. Some of the underlying principles of Buddhism are surprisingly relevant to the modern world. One of these is that the void, or nothingness,

is the source of all things. A second principle is that the universe and everything in it are composed of millions of small particles. The third law, included here, is that all things composed of particles or component parts are fated to encounter entropy and decay over time. Although this law was stated thousands of years before computers, it is certainly true of computer software: software decays and loses value over time. Constant maintenance over time can delay software entropy, as we see with aging legacy applications. But eventually all software systems will decay to the point of being withdrawn. See also the Lehman/Belady laws later in this paper, which are similar to Buddha's laws. It is interesting that Steve Jobs, former CEO of Apple Inc., became a Buddhist, in part because of its relevance.

Conway's Law

- Any piece of software reflects the organizational structure that produced it.

Empirical data tends to support this law. An additional caveat is that the size of each software component will be designed to match the team size that is assigned to work on it. Since many teams contain eight people, this means that even very large systems might be decomposed into components assigned to eight-person departments, which may not be optimal for the overall architecture of the application.

Crosby's Law

- Quality is free.

Empirical data supports Phil Crosby's famous law for software as well as for manufactured products. For software, high quality is associated with shorter schedules and lower costs than similar projects with poor quality. Phil Crosby was an ITT vice president who later became a global quality consultant. His book "Quality is Free" is a best-seller.

Gack's Law

- When executives or clients demand unrealistic and unobtainable project schedules, the probability of substantial cost overruns and schedule delays will double; the actual project's schedule will probably be twice the optimistic schedule demanded by the stakeholder.

This law has been known for many years by software quality and process consultants. However, in spite of hundreds of projects that end up in trouble, impossible schedules without the benefit of either accurate parametric estimates or accurate benchmarks from similar projects continue to be the most common way of developing medium to large applications between 1,000 and 10,000 function points in size. This size range is characterized by amateurish manual estimates and failure to bring in external benchmarks from similar projects. (Really large projects in the 100,000-function point size range tend to use professional estimating personnel, parametric estimating tools, and historical benchmark data, although many of these massive projects also get into trouble.)

Galorath's Seventh Law

- Projects that get behind stay behind.

Dan Galorath has a number of other laws, but this one has poignant truth that makes it among the most universal of all software laws. While there are some consultants who are turnaround specialists, by and large, projects that fall behind are extremely difficult to recover. Deferring features is the most common solution. Many attempts to recover lost time, such as skipping inspections or truncating testing, backfire and cause even more delays. This law is somewhat congruent with Brooks' Law, cited earlier. See also Gack's Law.

Gresham's Law

- Bad drives out good.

This law predates software and is named after a Tudor-era financier, Sir Thomas Gresham. The law was first stated for currency and refers to the fact that if two currencies are of unequal intrinsic value, such as gold and paper, people will hoard the valuable currency and drive it out of circulation. However, the law also has social implications. Studies of software engineer exit interviews reveal that software engineers with the highest appraisal scores leave jobs more frequently than those with lower scores. Their most common reason for leaving is "I don't like working for bad management." Restated for software sociological purposes, this law becomes "Bad managers drive out good software engineers."

Hartree's Law

- Once a software project starts, the schedule until it is completed is a constant.

Empirical data supports this law for average or inept projects that are poorly planned. For projects that use early risk analysis and have top teams combined with effective methods, this law is not valid. It applies to about 90 percent of projects, but not the top 10 percent. See also Brooks' Law, Gack's Law, and Galorath's Seventh Law.

Hick's Law

- The time needed to make a decision is a function of the number of possible choices.

This law was not originally stated for software, but empirical data supports this law for decisions regarding requirements issues, design issues, coding issues, and quality control issues. This law is related to complexity theory.

Humphrey's Law

- Users do not know what they want a software system to do until they see it working.

This law by the late Watts Humphrey is supported by empirical data for thousands of custom applications developed for external clients. However, inventors who build applications for their own use already have a vision of what the application is supposed to do. This law supports the concept of increments, each of which is usable in its own right. However, that is difficult to accomplish for large and complex applications.

Jevons' Law

- Increased efficiency in using a consumable product increases the demand for the product.

This law originated in 1865 when William Stanley Jevons noted that increased efficiency in burning coal had increased demand for that product. Although the law applied to a physical product, the same concept has been noted for computer memory chips and thumb drives • the better they are, the more we use them.

Jones' Law of Software Failures

- The probability of a software project failing and not being completed is proportional to the cube root of the size of the software application using IFPUG function points with the results expressed as a percentage. For 1,000 function points, the odds are about 8 percent; for 10,000 function points the odds are about 16 percent; for 100,000 function points the odds are about 32 percent.

This law is supported by empirical data from approximately 26,000 projects. However, government projects and information systems fail more frequently than systems software and embedded applications.

Jones' Law of Defect Removal Efficiency (DRE)

- Every form of defect removal activity has a characteristic efficiency level, or percentage of bugs actually detected. Most forms of testing are about 35 percent efficient, or find one code bug out of three. Inspections are about 85 percent efficient for all defect sources. Static analysis is about 55 percent for code bugs.

The metric of defect removal efficiency (DRE) was first developed by IBM in the early 1970s while IBM was exploring formal inspections as a method of improving overall software quality. There are two common ways of measuring DRE as of 2014. The original way used by IBM, Namcook Analytics, and many other companies is to measure internal bugs and compare these against bugs reported by users in the first 90 days of usage • if developers found 900 bugs and users reported 100 bugs in the first three months, the DRE is 90 percent. Another way was adopted by the International Software Benchmark Standards Group (ISBSG), which compares development defects against user-reported bugs found in the first 30 days of usage. The ISBSG results are usually about 15 percent higher in DRE than the original IBM method. The current U.S. average for DRE using the IBM and Namcook method is below 90 percent, but the best projects top 99 percent. The combination of function point metrics for defect density and defect removal efficiency (DRE) provides a very good method for quality analysis. By contrast, the "cost per defect" metric is harmful because it penalizes quality and is cheapest for the buggiest software. The software industry has very poor measurement practices and continues to use metrics such as "lines of code" and "cost per defect" that violate standard economic assumptions.

Jones' Law of Software Test Case Volumes to Achieve 98 Percent Test Coverage

- Raise application size in IFPUG function points to the 1.2 power to predict the probable number of test cases needed to achieve 98 percent test coverage for code paths and explicit requirements. Thus, for 100 function points there may be 251 test cases; for 1,000 function points there may be 3.981 test cases; for 10,000 function points there may be 63.095 test cases.

There are about 25 different kinds of testing for software, although the six most common forms of testing are 1) unit test, 2) new function test, 3) regression test, 4) component test, 5) system test and 6) beta test. The law stated above applies to the first five • beta tests are carried out by sometimes hundreds of external customers who all may test in different fashions. This law is based on empirical data from companies such as IBM and ITT, which use certified test personnel. Companies and projects where developers and amateurs perform testing would have a lower exponent and also lower test coverage. This law needs to be studied at frequent intervals. It would be useful to expand the literature on test case volumes and test coverage. Needless to say, cyclomatic complexity can shift the exponent in either direction.

Jones' Law of Software Development Schedules

- Raising application size in IFPUG function points to the 0.38 power provides a useful approximation of development schedules in calendar months. For 100 function points, the schedule would be about 5.8 months; for 1,000 function points the schedule would be about 13.8 calendar months; for 10,000 function points the schedule would be about 33.2 months.

This law is supported by empirical data from about 26,000 software projects. However, military and defense projects need a different exponent of about 0.4. Smaller Agile projects need a different exponent of about 0.36. Projects constructed primarily from reusable components need a different exponent of about 0.33.

Lehman/Belady Laws of Software Evolution

- Software must be continually updated or it becomes less and less useful.
- Software entropy or complexity increases over time.
- Software applications grow larger over time.
- Software quality declines over time.
- All users and software personnel must keep up-to-date with software changes.

These laws by Dr. Meir Lehman and Dr. Laszlo Belady of IBM were derived from a long-range study of IBM's OS/360 operating system. However, they have been independently confirmed by the author of this report and by other studies. The first law is obvious, but the second law is not. The continual modification of software to fix bugs and make small enhancements tends to increase cyclomatic complexity over time and thus increase the entropy or disorder of the software. In turn, this slows maintenance work and may require additional maintenance personnel unless replacement or restructuring occurs. Software renovation and restructuring can reverse entropy, or at least slow it down. See also Buddha's Third Law earlier in this document.

Love's Law of Legacy Application Architecture Changes

- If you want to modify the architecture of a legacy system, reorganize and restructure the support organization first and then wait a while.

This law is congruent with several other laws that observe that software architecture tends to reflect human organization structures, whether or not this is the best architecture for the software itself. This law is congruent with Conway's Law discussed earlier. There seems to be a fundamental truth in the observation that software mirrors human organizations, for good or for ill; probably for ill.

Love/Putnam Law of Maximum Schedule Compression

- Software project schedules have a fixed point of maximum compressibility. Once that point is reached, schedules can no longer be shortened, no matter how many or what kinds of resources are applied.

This law by Tom Love and Larry Putnam is an abstract version of the Jones law that shows IFPUG function points raised to the 0.38 power predict average schedules in calendar months. In general, the point of maximum compressibility is no more than about 0.3 below the average value; that is, if a 0.38 exponent yields an average schedule, a 0.35 exponent would yield the point below which schedules are no longer compressible. For 1,000 function points, a value of 0.38 yields 13.8 calendar months. A value of 0.35 yields 11.2 calendar months, beyond which further compression is not possible. A caveat is that constructing applications from libraries of certified reusable materials or using a requirements-model-based generator have both been shown to go past the point of incompressibility. Love's Law works for custom designs and hand coding, but not for mashups or applications built from standard reusable materials where manual coding is minimized or not used at all. The first version of this law was noted by the author of this paper in 1973 while building IBM's first parametric estimation tool. This is probably a case of independent discovery since Putnam, Love, and Jones were all looking at similar kinds of data.

Metcalfe's Law

- The value of a network system grows as the square of the number of users of the system.

This law is outside of the author's scope of research and the author's collection of data. It seems reasonable, but due to a lack of data, the author cannot confirm or challenge it here. It seems obvious that network value increases as more people use it, assuming high usage does not degrade performance and reliability.

Moore's Laws

- The power of computers per unit of cost doubles every 24 months.
- The number of transistors that can be placed on an integrated circuit doubles every 18 months.

These laws have been a mainstay of computing economics for many years. One by one, the law reaches the end point of various technologies, such as silicon and gallium arsenide, only to continue to work with newer technologies. Quantum computing is probably the ultimate end point at which the law will no longer be valid. However, Moore's laws have had a long and successful run • probably longer than most of the laws in this paper.

Murphy's Law

- If something can go wrong or fail, it will.

This is not a software law, but it is one that applies to all human constructions. Empirical data supports this law to a certain degree. The law is hard to study because some failures do not occur until years after software has been released and is in use. There is an interesting website that lists dozens of variations of Murphy's Laws applied to computer software: murphys-laws.com.

Paul's Principle

- Knowledge workers become less competent over time, since knowledge changes faster than practitioners learn new skills.

This is a thought-provoking observation for software specialists such as testers, business analysts, architects and the like. The concept seems to be supported by observations and evidence. It can be extended to corporations, since the rates of initial innovations in companies such as Apple and Microsoft slow down over time. Some kinds of knowledge work, such as medicine and law, have managed to overcome this principle by requiring continual education in order to keep licenses valid. Since software has no licenses and little required continuing education for professionals (as of 2016), this seems to be a weakness for software engineering.

Parkinson's Law

- Work expands to fill the time available for completion.

Software is labor intensive, and there is no strong supporting evidence of software engineers puffing up projects to fill vacant time since most software projects have very little vacant time available.

Senge's Law

- Faster is slower.

Peter Senge noted that, for business in general, attempts to speed up delivery of a project often made it slower. This phenomenon is true for software. Common mistakes made when trying to speed up projects include omitting inspections and truncating testing. These tend to stretch out software development, not shorten it. Hasty collection and review of requirements, jumping into coding prior to design, and ignoring serious problems are all practices that backfire and make projects slower. To optimize software development speed, quality control (including inspections and static analysis prior to testing) is valuable.

Pareto Principle

(Applied to Software Quality by Capers Jones)

- More than 80 percent of software bugs will be found in less than 20 percent of software modules.

The discovery of error-prone modules (EPM), which receive far more bug reports than normal, was first made in IBM in the 1970s and confirmed by other companies including ITT, AT&T and many others. In general, bugs are not randomly distributed but clump in a small number of modules, often with high cyclomatic complexity. This phenomenon is common on large applications above 1,000 function points in size. For the IBM IMS database project, about 57 percent of customer-reported bugs were found in 32 modules out of a total of 425 modules in the application. More than 300 IMS modules had zero-defect bug reports from customers. Inspections and surgical removal of error-prone modules raised IMS reliability and customer satisfaction at the same time that maintenance costs were reduced by more than 45 percent and development cycles were reduced by 15 percent. Such findings confirm Crosby's Law that software quality is indeed free. It often happens that less than five percent of software modules contain more than 95 percent of software bugs. The Pareto Principle has been explored by many software researchers, including Gerald Weinberg and Walker Royce, and it seems relevant to a wide range of software phenomena.

The Peter Principle

- In a hierarchy, every employee tends to rise to the level of his or her incompetence.

This is not an exclusively software observation but is a general business observation. It does not seem to hold for software technical work, since good software engineers may not have a level of incompetence. The law seems more relevant to subjective tasks than to engineering tasks. If the law is restricted to a management population rather than a population of technical personnel, it seems to have more relevance. Indeed, the most visible manifestations of this law are often at the CEO and corporate chair levels.

Weinberg's First Law

- If a program does not have to be correct, it can meet any other requirement.

This law is intriguing. Most programs are not correct, yet they are deployed and used daily. Only when serious bugs occur does the lack of correctness have a major impact. The essence of the idea is that correctness is difficult, but other factors are not as difficult.

Weinberg's Second Law

- If builders built buildings the way programmers write programs, a woodpecker could destroy civilization.

This law is the most thought-provoking law in this paper. It deserves serious consideration. Empirical data supports this law to a certain degree. Software applications with questionable architecture and high levels of cyclomatic and essential complexity are fragile. Small errors and even one line of bad code can stop the application completely or create large and expensive problems.

Weinberg/Okimoto Law of "TEMP" Hazards

- Any application that contains the string "TEMP" will be difficult to maintain because that string indicates temporary work that probably was done carelessly.

This interesting law by Jerry Weinberg and Gary Okimoto is derived from examining actual code strings in software. Those highlighted by markers indicating temporary routines have a tendency to become error prone.

Weinberg/Jones Law of Error-Prone Module (EPM) Causation

- A majority of error-prone modules (EPM) bypass some or all of proven effective quality steps such as inspections, static analysis, and formal testing.

This law was derived independently by Jerry Weinberg and the author from examination of error-prone modules (EPM) in different applications and in different development laboratories in different parts of the country. We both noted that a majority of error-prone modules had not followed proven and effective quality control methods such as inspections, static analysis, and formal testing. Root cause analysis also indicated that some of the careless development was due to the modules arriving late because of creeping user requirements.

Wirth's Law

- Software performance gets slower faster than hardware speed gets faster.

This law was stated during the days of mainframes and seemed to work for them. However, for networked microprocessors and parallel computing, the law does not seem to hold.

Yannis' Law

- Programming productivity doubles every six years.

The author's own data shows that programming productivity resembles a drunkard's walk, in part because application sizes keep getting larger. However, if you strip out require-

ments and design and concentrate only on pure coding tasks, then the law is probably close to being accurate. Certainly modern languages such as Java, Ruby, Go, C# and the like have better coding performance than older languages, such as Assembly and C. There is a caveat, however. Actual coding speed is not the main factor. The main factor is that modern languages require less unique code for a given application, due in part to more reusable features. Yannis' Law would be better if it specified separate results by application size and by application type. For example, there is strong evidence of productivity gains below 1,000 function points in size but little or no evidence for productivity gains above 10,000 function points. Productivity rates vary in response to team experience, methodologies, programming languages, CMMI levels, and volumes of certified reusable materials. For any given size and type of software project, productivity rates vary by at least 200 percent in either direction from the nominal average.

Zipf's Law

- In natural language, the frequency of a word is inversely proportional to its rank in the frequency table (that is, the most common word is used about twice as much as the second most common word). Zipf's Law appears to work with programming keywords as well as natural language text.

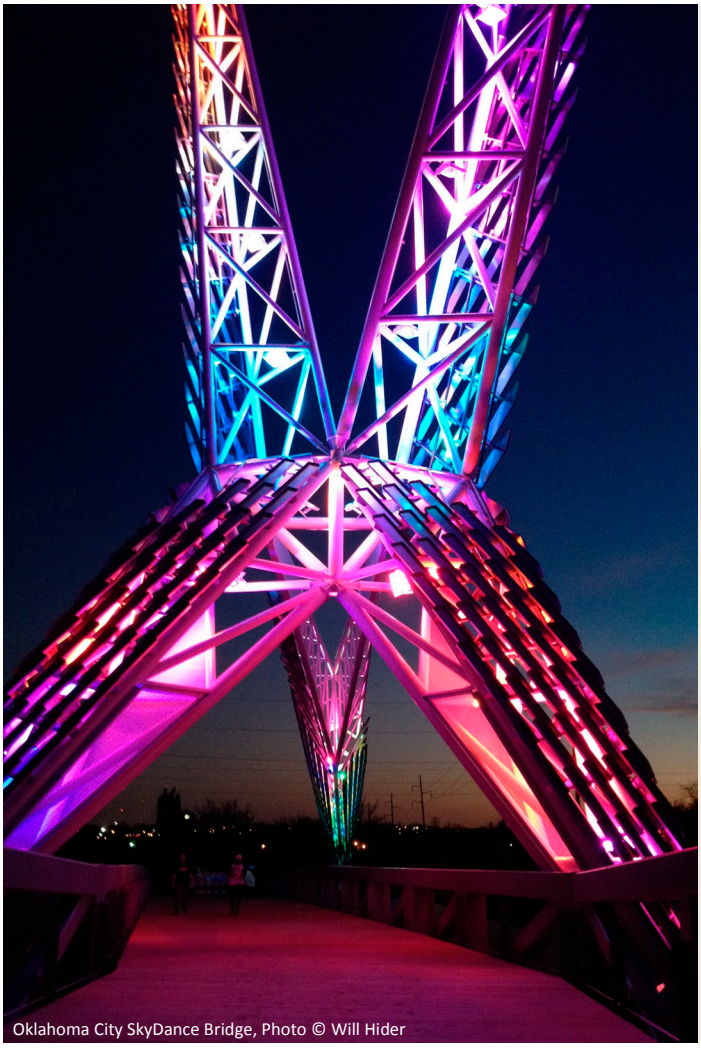
This law by George Zipf was originally developed based on linguistics patterns of natural languages long before software even existed. However, it does seem relevant to software artifacts, including requirements, design, and source code. A useful extension to Zipf's Law would be to produce a frequency analysis of the vocabulary used to define programs and systems as a step toward increasing the volume of reusable materials.

Summary and Conclusions

This list of software laws shows a number of underlying concepts associated with software engineering. The laws by the author were originally published over a 35-year period in 16 books and approximately 100 journal articles. This is the first time the author's laws have been listed in the same document.

These laws are derived from the author's collection of quantitative data, which started at IBM in 1970 and has continued to the current day. The author was fortunate to have access to internal data at IBM, ITT, and many other major software companies. The author has also had access to data while working as an expert witness in a number of software lawsuits.

While many laws are included in this article, no doubt many other laws are missing. This is a work in progress, and new laws will be added from time to time.



Oklahoma City SkyDance Bridge, Photo © Will Hider

REFERENCES

Note: A Google search on phrases such as "software laws" and "software engineering laws" will return a variety of interesting sources. The references included here are only a small portion of the available literature.

Boehm, Barry. (1981.) "Software Engineering Economics." Prentice Hall, Englewood Cliffs, N.J.

Brooks, Fred. (1974, rev. 1995.) "The Mythical Man-Month." Addison-Wesley, Reading, Mass.

Campbell-Kelly, Martin. (2003.) "A History of the Software Industry: from Airline Reservations to Sonic the Hedgehog." The MIT Press, Cambridge, Mass. ISBN 0-262-03303-8. 372 pages.

Crosby, Philip B. (1979.) "Quality is Free." New American Library. Mentor Books. New York, N.Y. 270 pages.

DeMarco, Tom (1999.) "Peopleware: Productive Projects and Teams." Dorset House. New York, N.Y. ISBN 10: 0932633439. 245 pages.

DeMarco, Tom & Lister, Tim. (2003.) "Waltzing with Bears: Managing Risks on Software Projects." Dorset House Press, N.Y.

Gack, Gary. (2010.) "Managing the Black Hole – The Executive's Guide to Project Risk." The Business Expert Publisher. Thomson, Georgia. ISBSG10: 1-935602-01-2.

Humphrey, Watts. (1989.) "Managing the Software Process." Addison Wesley. Reading, Mass.

Jones, Capers & Bonsignour, Olivier. (2011.) "The Economics of Software Quality." Addison Wesley Longman. Boston, Mass. ISBN 10: 0-13-258220–1. 585 pages.

Jones, Capers. (2014.) "The Technical and Social History of Software Engineering." Addison Wesley.

Jones, Capers. (2010.) "Software Engineering Best Practices." McGraw Hill. New York, N.Y. ISBN 978-0-07-162161-8. 660 pages.

Jones, Capers. (2007.) "Estimating Software Costs." McGraw Hill. New York, N.Y. ISBN 13-978-0-07-148300-1.

Jones, Capers. (1994.) "Assessment and Control of Software Risks." Prentice Hall. ISBN 0-13-741406-4. 711 pages.

Jones, Capers. (December 1995.) "Patterns of Software System Failure and Success." International Thomson Computer Press. Boston, Mass. 250 pages. ISBN 1-850-32804-8. 292 pages.

Jones, Capers. (2000.) "Software Assessments, Benchmarks, and Best Practices." Addison Wesley Longman. Boston, Mass. ISBN 0-201-48542-7. 657 pages.

Jones, Capers. (January 1977.) "Program Quality and Programmer Productivity." IBM Technical Report TR 02.764, IBM. San Jose, Calif.

Kan, Stephen H. (2003.) "Metrics and Models in Software Quality Engineering, 2nd edition." Addison Wesley Longman. Boston, Mass. ISBN 0-201-72915-6. 528 pages.

Kuhn, Thomas. (1996.) "The Structure of Scientific Revolutions." University of Chicago Press. Chicago, Ill. ISBN 0-22645807-5. 212 pages.

Love, Tom. (1993.) "Object Lessons: Lessons Learned in Object-Oriented Development Projects." SIGS books. ISBN 0-9627477-3-4. 266 pages.

McConnell, Steve. (1997.) "Software Project Survival Guide." Microsoft Press.

Pressman, Roger. (2005.) "Software Engineering – A Practitioner's Approach." McGraw Hill. N.Y. 6th edition. ISBN 0-07-285318-2.

Radice, Ronald A. (2002.) "High Quality Low Cost Software Inspections." Paradoxicon Publishing. Andover, Mass. ISBN 0-9645913-1-6. 479 pages.

Starr, Paul. (1982.) "The Social Transformation of American Medicine." Basic Books. Perseus Group. ISBN 0-465-07834-2. NOTE: This book won a Pulitzer Prize in 1982 and is highly recommended as a guide for improving both professional education and professional status. There is much of value for the software community.

Strassmann, Paul. (1985.) "Information Payoff." Information Economics Press. Stamford, Conn.

Strassmann, Paul. (2004.) "Governance of Information Management: The Concept of an Information Constitution." 2nd edition (eBook). Information Economics Press. Stamford, Conn.

Strassmann, Paul. (1999.) "Information Productivity." Information Economics Press. Stamford, Conn.

Wiegers, Karl E. (2002.) "Peer Reviews in Software – A Practical Guide." Addison Wesley Longman. Boston, Mass. ISBN 0-201-73485-0. 232 pages.

Weinberg, Gerald M. (1971.) "The Psychology of Computer Programming." Van Nostrand Reinhold, New York. ISBN 0-442-29264-3. 288 pages.

Weinberg, Gerald M. (1986.) "Becoming a Technical Leader." Dorset House. New York. ISBN 0-932633-02-1. 284 pages.

Yourdon, Ed. (1997.) "Death March - The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects." Prentice Hall PTR. Upper Saddle River, N.J. ISBN 0-13-748310-4. 218 pages.

ABOUT THE AUTHOR

Capers Jones is currently vice president and chief technology officer of Namcook Analytics LLC. Prior to the formation of Namcook Analytics in 2012, he was the president of Capers Jones & Associates LLC. He is the founder and former chairman of Software Productivity Research LLC (SPR). Capers Jones founded SPR in 1984 and sold the company to Artemis Management Systems in 1998. He was the chief scientist at Artemis until retiring from SPR in 2000.

Before founding SPR, Capers was Assistant Director of Programming Technology for the ITT Corporation at the Programming Technology Center. During his tenure, he designed three proprietary software cost and quality estimation tools for ITT between 1979 and 1983. He was also a manager and software researcher at IBM in California where he designed IBM's first two software cost estimating tools in 1973 and 1974 in collaboration with Dr. Charles Turk. Capers Jones is a well-known author and international public speaker. Some of his books have been translated into five languages. His most recent book is *The Technical and Social History of Software Engineering*, Addison Wesley 2014.

Capers Jones has also worked as an expert witness in 15 lawsuits involving breach of contract and software taxation issues and provided background data to approximately 50 other cases for other testifying experts.

Capers.Jones3@gmail.com
www.Namcook.com