Introduction

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.

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 in that for projects with varying team sizes 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 $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% 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.

Brook’s 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 goes up with application size and team size. The larger the application the more difficult it is to recover from schedule delays by any means. For small projects with less than 5 team members adding one more experienced person will not stretch the schedule but adding a novice will. For large applications with more than 100 team members these projects almost always run late due to poor quality control and poor change control. Adding people tends to slow things down due to training and complex communication channels.

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 8 people this means that even very large systems might be decomposed into components assigned to 8-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.

Deming’s 14 principles

1. Create a constant drive towards improvement
2. Adopt a new philosophy
3. Stop depending on post development inspections
4. End the practice of lowest-cost contracts
5. Improve constantly and forever
6. Use on-the-job training
7. Institute leadership
8. Drive out fear so everyone can contribute
9. Break down barriers between departments
10. Eliminate slogans and targets
11. Remove barriers to pride of workmanship for workers
12. Remove barriers to pride of workmanship for managers and engineers
13. Institute vigorous training and education
14. Let everyone contribute

Empirical data from many industries including software support 13 of Deming’s 14 principles. The only principle not supported for software is “don’t depend upon inspections.” For software inspections serve as both a defect prevention and defect removal method and are performed before the software is finished when there is still time to make changes. For hardware inspections are performed after the fact when the product is already built and hence expensive to modify. Deming, Jurand, and Crosby did not work in software but their general observations on the economics of quality and what is needed to achieve quality are all relevant to software engineering.

Ellison’s law of data consolidation
- Data that is integrated and centralized is more valuable than the sum of the individual pieces of data.

Empirical data supports this law. Integrated data bases are more valuable than their separate data elements. However in today's world of cyber attacks integrated data bases are also prone to hacking and theft. One major problem with data economic analysis is the lack of an effective size metric such as a “data point” metric that uses logic and counting rules similar to function point metrics. Without an effective size metric for data, neither economic value nor costs of data ownership can be evaluated, and this is also true for data quality.

Fitt’s law

- The time to acquire a target is a function of the distance to and the size of the target.

This law is not about software and applies to many physical phenomena. However it has a certain relevance to the human factors of using software and also using computers. Commands or control buttons that are large and conveniently located speeds up use. Microsoft’s tendency to hide basic commands such as putting the power-off command off screen in Windows 8 tends to slow down user speed. Some android smart phone features and functions are also hidden in unusual places.

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 1000 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 use historical benchmark data although many of these massive projects also get into trouble.)

Galorath’s 7th 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 turn-around 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 Brook’s law, cited earlier. See also Gack’s law.

Gresham’s law

- Bad drives out good

This law pre-dates 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. From studies of software exit interviews it has been noted that software engineers with the highest appraisal scores leave jobs more frequently than those with lower scores, and their most common reason for leaving is “I don’t like working for bad management.” Restated for software sociological purposes, “bad managers drive out good software engineers.”

Grosch’s law

- The cost of a computing system increases as the square root of the computational power of a computing system.

Due to the low costs and increasing power of micro computers this law does not seem to be valid in 2014. Parallel networks of linked computers are both powerful and inexpensive. The law worked during the mainframe era which is when it was originally stated.

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 works for about 90% of projects but not for the top 10%. See also Brook’s la, Gack’s law, and Galorath’s 7th 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 for applications built by inventors for their own use they 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.

Jones’s 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%; for 10,000 function points the odds are about 16%; for 100,000 function points the odds are about 32%.

This law is supported by empirical data from around 20,000 projects. However government projects and information systems fail more frequently than systems software and embedded applications.

Jones’s law of software failure culpability
• Software managers cause more project failures than software engineers.

*From working as an expert witness in a number of software lawsuits for projects that either failed and did not get delivered at all or which did not work properly when delivered, it was clear that project managers cause most of the failures. Bad decisions such as skipping inspections and truncating testing are major causes of failure. Worse, project managers have a tendency to conceal problems from both clients and senior executives until it is too late to fix them. Depositions and discovery documents highlighted the fact that software engineers knew about the problems and reported them to managers, but the managers failed to take effective actions and also failed to include the problems in status reports. The overall set of factors that cause software breach of contract litigation include poor estimates before starting, poor risk analysis before starting, poor quality control, poor change control, and poor status tracking.*

**Jones’s law of software quality costs**

• Cost per defect stays flat from function test through maintenance. Cost per defect penalizes quality and is lowest for the buggiest software. The best metric for quality costs and quality economic study is “defect removal costs per function point.”

*For more than 50 years the software community has lived by the aphorism that “it costs 100 times as much to fix a bug after release as early.” This is not true. Actual costs per defect stay flat from function test through maintenance. What happens is that the fixed costs associated with defect repairs become a higher percentage of total costs. Here are two examples: case 1 for fixing 50 bugs and case 2 for fixing 10 bugs. Assume that in both cases the software is 100 function points in size. In both cases test preparation is $10,000, test execution is $10,000, and it costs exactly $500 for each actual bug repair. For case 1 with 50 bugs the total costs were $45,000 for testing and repairs. For case 2 with 10 bugs the total costs $25,000 for testing and repairs. In case 1 the cost per defect is $900 but in case 2 the cost per defect is up to $2,500 due to the fixed costs of preparation and execution. In case 1 the defect removal cost per function point is $450 but in case 2 the defect removal cost per function point is down to $250. As can clearly be seen cost per defect violates standard economic principles and penalizes quality while defect removal cost per function point is congruent with standard manufacturing economics and shows the economic value of quality.*

**Jones’s law of historical data “leakage”**

• What is called “historical data” leaks and omits more than 50% of true projects costs for more than 80% of all software applications.
Software historical data suffers from chronic “leakage” or leaving important activities and cost drivers. For information technology projects what is called historical data only includes about 37% of total costs. Common omissions include unpaid overtime, management, and the work of part-time specialists such as software quality assurance, technical writers, function point counters, project office personnel, and user costs. Some companies only measure design, code, and unit test (DCUT) which comprises less than 30% of total costs. Quality data leaks too and is even less complete. Common omissions include all forms of defect removal prior to function testing; i.e. inspections, static analysis, and unit test are not measured for either defect removal efficiency or costs. It is difficult to validate the accuracy of software cost estimates using historical that itself has not been validated. A few companies such as IBM have accurate data. Projects created under time and materials contracts tend to have good data in order to get paid. This topic is seldom covered in the software literature, which is somewhat surprising. The hard sciences have very accurate data for most phenomena, but software engineering has been content with very inaccurate data without even knowing it.

Jones’s law of creeping requirements

- Between the end of the requirements phase and the end of the coding phase software requirements will grow and creep at measured rates of between 1% and 2% per calendar month.

This law is supported by over 1,000 projects where function points were counted at the end of requirements and then again at delivery. If an application was 1000 function points at the end of requirements and 1,120 at the end of the coding phase the net growth is 120 function points. Assuming the schedule between requirements end and coding end was 10 months the net growth per month 12 function points or 1.2% per calendar month. A few measured projects have topped 4% per calendar month in requirements creep. The larger the application the greater the total creep due to much longer schedules. For very large systems total creep can top 50% of final application size at delivery. There is nothing wrong with function point creep: it should be expected. Agile project expect change and are geared to handle them. In fact the author’s Software Risk Master (SRM) tool predicts requirements creep during development and also for three years after delivery. The problem is that requirements creep is often omitted from initial cost estimates and hence degrades the value of software by making costs larger than anticipated.

Jones’s law of post-release application growth

- Once software applications are released to users they will continue to grow at rates between 8% and 15% per calendar year so long as the software has active users.
This law is based on more than 500 projects where growth records exist for five or more calendar years after release. The best data on post-release growth comes from major companies such as IBM which keep longitudinal records for every software project for many years. The growth is not always smooth. About every four years commercial vendors tend to add “mid life kickers” to keep software competitive. Post release growth is a common phenomenon that causes trouble with benchmarks. Since application size keeps changing it is necessary to recalculate benchmarks every calendar year or every fiscal year.

**Jones’s law of software defect creation**

- Raising software application size in IFPUG function points to the 1.1 power will give a useful approximation of total bugs that will be found in requirements, design, code, user documents, and bad fixes. For 1,000 function points the defect potential will be about 1,952 bugs; for 10,000 function points the defect potential will be about 25,118 bugs.

Software bugs or defects are found in requirements, design, code, user documents, and also “bad fixes” or secondary bugs in defect repairs themselves. Empirical data supports this law for projects between 10 and 50,000 function points although there are wide ranges based on client and team experience, methodologies, programming languages, and CMMI levels. Earlier versions of this law from the 1980’s used an exponent of 1.2 power due to lower-level programming languages. This law provides a reasonably good approximation of defects in all sources – not just coding defects.

**Jones’s law of software defect removal economics**

- Projects that are above 95% in defect removal efficiency are faster and cheaper than projects that are below 90% in defect removal efficiency.

Empirical data from about 20,000 projects supports this law. Projects that use only testing are normally below 90% in defect removal efficiency (DRE) and usually run late due to stretching out the test interval. Similar projects that use inspections and static analysis before testing can top 99% in DRE and also are usually on time or early, assuming rational schedule planning in the first place. Poor defect removal efficiency (DRE) is the main reason for schedule slippage. Attempts to shorten schedules by bypassing inspections and truncating testing usually lengthen the schedules. As several other laws such as Crosby’s demonstrate, high quality is not expensive. It is poor quality that drives up software costs.
Jones’s 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% efficient or find one code bug out of three. Inspections are about 85% efficient for all defect sources. Static analysis is about 55% for code bugs.

The metric of defect removal efficiency (DRE) was first developed in IBM in the early 1970’s at the time that IBM was exploring formal inspections as a method of improving overall software quality. There are two common ways of measuring DRE in 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; i.e. if developers found 900 bugs and users reported 100 bugs in the first three months then DRE is 90%. 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% higher in DRE than the original IBM method. The current U.S. average for DRE using the IBM and Namcook method is below 90% but the best projects top 99%. The combination of function point metrics for defect density normalization combined with defect removal efficiency (DRE) provides a very good method for quality analysis. By contrast the “cost per defect” metric is harmful since 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’s law of bad-fix injections

- On average about 7% of software bug repairs will have a new bug in the repair itself. These were first identified by IBM and are called “bad fixes.”

This law is based on empirical data from both internal defect repairs and also from data taken from the discovery and depositions of software lawsuits for poor quality. Defect repairs are themselves significant sources of new errors. The solution is to improve quality control for defect repairs via static analysis, inspections, or other methods. Sometimes bad fix injections have topped 25% of for error-prone modules with high cyclomatic complexity levels. Bad fix injections can be minimized by using static analysis and regression testing on all defect repairs and by using formal inspections on major defect repairs. In one lawsuit where the author was an expert witness the vendor tried four times to fix a bug and each fix not only failed but added new bugs. It was not until the fifth attempt, nine months after the bug was reported, that the vendor successfully repaired the original bug without adding new bugs.
Jones’s law of bad test cases

- There may be as many bugs in test cases as there are in the software being tested. Running bad test cases raises testing costs but lowers testing defect removal efficiency.

One of the least studied topics in all of the software quality literature is that of bugs or defects in test cases and test scripts themselves. Both test cases and test scripts are complex and difficult to create and hence both are prone to errors. IBM research that used inspections on test plans, test cases, and test scripts found almost as many bugs in test materials as in the software being tested. This topic is in urgent need of modern studies. One thing that would minimize test errors would be the availability of suites of certified reusable test materials and even reusable test cases for common types of software such as banking applications, insurance applications, telecommunications applications, and others that are widely used and similar across many companies. Custom designs and custom code are intrinsically error prone and so are custom unique test cases. The volume of errors and mistakes in test plans, test cases, and test scripts is in urgent need of better studies.

Jones’s law of software test case volumes to achieve 98% 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% test coverage for code paths and explicit requirements. Thus for 100 function points there may be 251 test cases; for 1000 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’s law of software development cost drivers

- Finding and fixing bugs is the most expensive activity in all of software. Producing paper documents is the second most expensive activity for software. Coding is number three. Meetings and communications are number four.

Empirical data from about 20,000 projects support this law. Finding and fixing bugs is the #1 cost driver for applications between 10 and 100,000 function points. Paperwork is the number two cost driver primarily for larger applications above 5,000 function points. For military and defense software larger than 10,000 function points the volume of paper documents is about three times larger than for similar civilian projects of the same size. In this case paperwork is the #1 cost driver for military software and finding and fixing bugs drops to the #2 cost driver. Below 100 function points the three top cost drivers are coding, defect removal, paperwork. Above 1000 function points the three top cost drivers are defect removal, coding, paperwork. For large military projects above 10,000 function points the three main cost drivers are paperwork, defect removal, coding. Some defense projects create more than 100 documents and more than 3,000 words for every line of code. Code is of course the deliverable that makes software useful, but the two other cost drivers of producing paper documents and finding and fixing bugs are what makes software expensive. Note that for agile projects the cost drivers are finding and fixing bugs, meetings and communications, programming, and producing paper documents. Agile has reduced paper volumes but increased face to face meetings. The basic point is that coding is only one of several important cost drivers, and therefore software economic analysis needs to include paperwork costs, defect removal costs, and meeting and communication costs as well as coding costs.

Jones’s 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 1000 function points the schedule would be about 13.8 calendar months.

This law is supported by empirical data from about 20,000 software projects. However military and defense needs 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.
Jones’s law of programming language creation

- New programming languages will be developed at rates of more than 2 per month forever.

This law was first stated in 1984 when the author began a survey of programming languages in order to predict the number of source code statements per function point. As of 2014 there are close to 3,000 named programming languages and new languages continue to be developed at what might be an accelerating rate. No one knows why this phenomenon occurs. No one knows why the software industry needs thousands of programming languages especially when the majority of applications are written in the most common languages. Probably sociological reasons are part of the constant creation of new languages.

Jones’s law of programming language utility 1

- There are no programming languages that are optimal for all known forms of software.

This law is supported by empirical data from around 20,000 software projects. Languages tend to fall into rough categories of those used for scientific and mathematical work; those used for data base and data-intensive applications; those used by systems and embedded applications; and several others. Attempts to build a universal language such as PL/I by IBM have not been successful in even slowing down the development of more specialized languages. No doubt some languages can be used for every type of software, just as no doubt an automobile could be used to plow a field. However as a general statement specialization seems to be the norm for programming languages.

Jones’s law of programming language utility 2

- Multiple programming languages are needed for all complex software systems. The range of observed languages in specific applications is from 2 to 15.

This observation is supported by empirical data from about 20,000 software projects. Almost the only applications coded in only one programming language are very small applications below 10 function points or 500 logical code statements in size. Most applications use at least two languages such as COBOL and SQL or Java and HTML. From the author’s data the average number of programming languages per application is about 2.5 but many applications use more than half a dozen. The largest number noted by the author in one application was 15 languages.
Jones’s law of programming language utility 3

- In every decade less than 10% of the available programming languages are used to code over 90% of all software applications created during the decade.

This law is supported by a stream of empirical data between the years 1965 and 2014. The popularity of programming languages is very transient and popularity seems to average less than a 3.5 year period from a burst of initial popularity until the language starts to fade away. Nobody knows why this phenomenon occurs. Some languages such as Objective C used by Apple have persistent use over many years. Why programming languages come and go is not fully understood, nor is language persistence. In any case some form of museum or repository for language materials that included working compilers and instructional materials as well as code samples would benefit the software maintenance community.

Jones’s law of programming language utility 4

- The average age of large software applications is older than the useful life of the programming languages used to code it; i.e. most large legacy systems were coded in dead or dying languages.

This law is supported by empirical data from more than 10,000 legacy applications whose ages range from more than 30 calendar years to about 3 calendar years. Some legacy applications are so old that working compilers no longer exist. For many legacy languages such as MUMPS, CORAL, CHILL, etc. there are very few young programmers who even know the language. In another report the author suggests the need for a national museum of programming languages that would include working compilers and code samples for all known programming languages, together with tutorial materials for teaching the languages to those tasked with maintain legacy software written in dead languages.

Jones’s law of software methodologies

- No specific software methodology is optimal for all sizes, classes, and types of software application. Large corporations building a variety of software project sizes, classes, and types will need at least agile, RUP, and TSP for development methods.

This law is supported by more than 20,000 applications developed in a total of 35 named development method between about 1965 and 2014. Methodology deployment and methodology selection tend to resemble cults more than technical decisions. There is very little validation of methodologies before release, and very little rational thinking about adopting methodologies. In general the most popular methods are adopted not for their technical merits but merely because
Pair programming is an example of a method released without validation and used without empirical data of success. Some of the approaches and methods that were validated and do have empirical data include agile development for small projects, the capability maturity model, formal inspections, Rational Unified Process (RUP), structured development, Team Software Process (TSP), test-driven development; Prince2, Merise, iterative development, automated requirements modeling, and some forms of application generation. Mashups or construction of new applications from pieces of old applications also has empirical data. By contrast cowboy development and waterfall development grew spontaneously and were not validated nor does empirical data show success.

Jones’s law of software occupation groups

- The usual number of software occupations for 1,000 function points is 4 (programmer; tester; technical writer; manager). The number of software occupations or specialists increase by 1 as software doubles in size (i.e. 5 occupations for 2000 function points; six occupations for 4000 function points, etc.) The industry total is about 126, distinct software occupations. Examples of these many occupations in alphabetical order include agile coaches, business analysts, configuration control specialists, customer support, function point analysts, human factors specialists, integration specialists, maintenance programmers, programmers (software engineers), project office specialists, project managers, software quality assurance, technical writers, and test specialists.

This law is supported by empirical data from around 20,000 software projects. It is also supported by a formal study of software occupation groups employed in major organizations. The study was funded by AT&T and included IBM, Texas Instruments, Ford Motors, the Navy, and a number of other large software employers. A related law not explicitly included is that specialists often but not always outperform generalists. This topic needs additional study. For example more research is needed on the performance of certified test personnel versus testing by developers.

Jones’s laws of software development staffing

- For top teams dividing application size in IFPUG function points by 175 gives a useful approximation of team size.
- For average teams dividing application size in IFPUG function points by 150 gives a useful approximation of team size.
- For novice teams dividing application size in IFPUG function points by 125 gives a useful approximation of team size.
The staffing values include all development personnel and project managers. These laws are supported by about 10,000 projects large enough to have development teams, but excluding small projects with teams of less than 3 people. For any given application size such as 1000 function points it takes fewer experts than it does average or novice personnel. The author’s Software Risk Master (SRM) tool predicts the team size based on experience levels of the various teams such as developers, testing, quality assurance, etc. SRM predicts a total of 20 occupation groups and includes both average and peak staff sizes.

Jones’s laws of software maintenance staffing

- For well-structured code and top teams dividing application size by 3,000 IFPUG function points gives a useful approximation of maintenance team size.
- For average code and average teams dividing application size by 1,500 IFPUG function points gives a useful approximation of maintenance team size.
- For poorly structured code and inexperienced teams dividing application size by 500 IFPUG function points gives a useful approximation of maintenance team size.

These laws are supported by more than 10,000 legacy applications undergoing maintenance between about 1975 and 2014. The law is based on the development of function point metrics inside IBM circa 1975. The older “lines of code” metric could also be used with the caveat that some languages are much easier to maintain than other languages. These predictions are for personnel tasked with finding and fixing bugs and keeping software operational. Pure customer support and enhancement specialists can be predicted, but need different algorithms. Overall there are 23 separate kinds of work subsumed under the generic name of “maintenance.” All 23 can be predicted, but doing so requires much more sophisticated methods than rules of thumb.

Jones’s law of software reusability

- It is theoretically possible to construct any arbitrary software system using 90% standard reusable components.

As of 2014 the average volume of reuse is only about 15% for the author’s overall collection of measured software projects. For certain types of software such as compilers and PBX switching software the volume of reuse does approach 90%. In order to expand certified reusable components to other kinds of software there is still work to be done in defining the specific functions that occur with high frequency across multiple application types. Because custom designs and manual coding are intrinsically expensive and error prone, reuse is the best overall long range strategy for software engineering.
Lehman/Belady laws of software evolution

- Software must be continuously updated or it becomes less and less useful.
- Software entropy or complexity increases over time.

These laws by Dr. Meir Lehman and Dr. Laszlo Belady of IBM were derived from 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 intuitively obvious but the second law is not. The continuous modification of software to fix bugs and make small enhancements tends to increase cyclomatic complexity over time and hence increase the entropy or disorder of the software. In turn, this slows maintenance work and may need additional maintenance personnel unless replacement or restructuring occur. Software renovation and restructuring can reverse entropy.

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 and what kinds of resources are applied.

This law by Larry Putnam and Tom Love is an abstract version of the Jones law that shows IFPUG function points raised to the 0.38 power predicts average schedules in calendar months. In general the point of maximum compressibility is no more than about .3 below the average value; i.e. if a 0.38 exponent yields and average schedule a 0.35 exponent would yield the point below which schedules are no longer compressible. For 1000 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 build 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 when building IBM’s first parametric estimation tool. Probably this is 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 lack of data by the author cannot be confirmed or challenged here.*

**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 point the law will no longer be valid. However Moore’s laws have had a long and very 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 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 web site that lists dozens of variations of Murphy’s laws applied to computer software; Murphys-laws.com.*

**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 for business in general that 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 and not shorten it. Hasty collection and review of requirements, jumping into coding prior to design, and ignoring serious problems are all topics that backfire and make projects slower. To optimize software development speed quality control including inspections and static analysis prior to testing are valuable.

Pareto principle applied to software quality by Capers Jones

• More than 80% of software bugs will be found in less than 20% 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 1970’s and then confirmed by other companies such as 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 data base project about 57% 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 over 45% and development cycles were reduced by 15%. Such findings confirm Crosby’s law that for software quality is indeed free. It often happens that less than 5% of software modules contain more than 95% of software bugs. The Pareto principle has been explored by many software researchers such as 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 a software observation but a general business observation. For software technical work it does not seem to hold, since good software engineers may not have a level of incompetence. The law seems more relevant to subjective tasks than to engineering tasks.

Weinberg’s 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) bypassed 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 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 carless 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 6 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 requirements 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 showed separate results by application size and by application type. For example there is strong evidence of productivity gains below 1000 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% 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 (i.e. the most common word is used about twice as much as the second most common word). Zipf’s law appears to work with programming key words 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 towards 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 author were originally published over a 35 year period in 16 books and about 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 in 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.

References and Readings

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


Crosby, Philip B.; Quality is Free; New American Library, Mentor Books, New York, NY; 1979; 270 pages.

DeMarco, Tom and Lister, Tim; Waltzing with Bears: Managing Risks on Software Projects; Dorset House Press, NY; 2003.


Gilb, Tom and Graham, Dorothy; Software Inspections; Addison Wesley, Reading, MA; 1993; ISBN 10: 0201631814.

Humphrey, Watts; Managing the Software Process; Addison Wesley, Reading, MA; 1989.

Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley Longman, Boston, MA; ISBN 10: 0-13-258220—1; 2011; 585 pages.

Jones, Capers; The Technical and Social History of Software Engineering; Addison Wesley, 2014.


Jones, Capers; Program Quality and Programmer Productivity; IBM Technical Report TR 02.764, IBM San Jose, CA; January 1977.


Love, Dr. Tom; *Object Lessons: Lessons Learned in Object-Oriented Development Projects*; SIGS books; 1993; ISBN 0-9627477-3-4; 266 pages.


Starr, Paul; *The Social Transformation of American Medicine*; Basic Books; Perseus Group; 1982; 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; *Information Payoff*; Information Economics Press, Stamford, Ct; 1985.

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

Strassmann, Paul; *Information Productivity*; Information Economics Press, Stamford, Ct; 1999.

